TypeScriptin hyödyllisyys JavaScript-ohjelmistokehityksessä

Jani Rapo

Pro gradu -tutkielma

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Helsinki, 18. helmikuuta 2020

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Jani Rapo

Työn nimi — Arbetets titel — Title

TypeScriptin hyödyllisyys JavaScript-ohjelmistokehityksessä

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	18. helmikuuta 2020	80 sivua + 5 sivua liitteissä

Tiivistelmä — Referat — Abstract

JavaScript on viime vuosina noussut yhdeksi suosituimmista ja käytetyimmistä ohjelmointikielistä, mutta sen toteutukseen ja ominaisuuksiin on tunnistettu liittyvän tiettyjä puutteita ja epäkohtia. Merkittävin näistä lienee dynaaminen tyypitys ja sen aiheuttamat automaattiset tyyppimuunnokset, jotka voivat johtaa väärien tyyppien tahattomaan käyttöön, ja sen myötä ohjelman odottamattomaan käyttäytymiseen. Puutteiden ja epäkohtien paikkaamiseksi on kehitetty erilaisia lisäosia ja työkaluja, joista TypeScript on viime aikoina noussut merkittävään asemaan.

TypeScript on Microsoftin luoma ja ylläpitämä ohjelmointikieli, joka tuo JavaScript-kieleen lisäominaisuuksia. Päällimmäinen näistä on tyyppijärjestelmä, jonka mahdollistama staattinen koodianalyysi ja tyyppitarkistus voivat auttaa selkeyttämään ohjelman käyttäytymistä eri tilanteissa ja mahdollisesti jopa vähentämään yksikkötestien tarvetta, kun koodissa tapahtuviin automaattisiin tyyppimuunnoksiin ei tarvitse varautua yhtä kattavasti.

Tutkielmassa tutkitaan, miten hyödyllisenä tai haitallisena TypeScriptiä pidetään web-ohjelmistoteollisuuden yrityksissä työskentelevien asiantuntijoiden keskuudessa, vastaavatko asiantuntijoiden mielipiteet aikaisempia tieteellisiä tutkimustuloksia ja millä ohjelmistokehitysprosessin osa-alueilla TypeScriptin hyödyt ja haitat korostuvat. Tutkielman tutkimusmenetelmänä on kyselytutkimus, joka toteutetaan Internetissä suoritettavan kyselylomakkeen avulla.

Tulokset osoittavat, että TypeScriptin avulla voidaan ratkaista useita JavaScriptiin liittyviä tunnistettuja puutteita. TypeScript tuo lisäksi mukanaan monia hyötyjä, jotka painottuvat ohjelmistokehitysprosessin eri osa-alueille. Hyödyistä päällimmäisinä nousevat esille staattinen koodianalyysi ja tyyppitarkistus, refaktoroinnin helppous, tyyppimerkinnät kooditason dokumentaationa, täsmälliset kehitystyökalujen automaattitäydennys- ja koodivihjeet sekä tarve vähemmille testeille. Tunnistettujen haittojen osalta TypeScript saattaa jopa hidastaa pienten projektien tekoa ja sen tyyppijärjestelmän opetteluun voi kulua aikaa. Tutkielman tulokset vahvistavat aikaisemmissa tieteellisissä tutkimuksissa tehtyjä havaintoja.

ACM Computing Classification System (CCS):
Software and its engineering → Software notations and tools → Software creation and management

Avainsanat — Nyckelord — Keywords
JavaScript, TypeScript, tyyppijärjestelmät, tyypitys, ohjelmistokehitys

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

JavaScript on viime vuosina noussut yhdeksi suosituimmista ja käytetyimmistä ohjelmointikielistä. Kieltä käytetään hyvin monipuolisesti, pääasiallisesti erilaisten web-sovellusten luontiin.

Nykypäivänä useimmat verkkosivut käyttävät JavaScriptiä, jonka tärkein tehtävä on lisätä verkkosivuille vuorovaikutteisuutta sekä dynaamista toiminnallisuutta. JavaScriptin käyttö ei kumminkaan rajoitu pelkästään selainpuolelle, vaan myös palvelinpuolen sovelluksia toteutetaan yhä enemmän JavaScriptin avulla. JavaScriptin käyttö on yleistynyt sen yksinkertaisuuden, joustavuuden, monipuolisten lisäosien ja tehokkaan suorituskyvyn ansiosta.

JavaScriptin toteutukseen ja ominaisuuksiin liittyen on kumminkin tunnistettu tiettyjä puutteita ja epäkohtia. Merkittävin näistä lienee dynaaminen tyypitys ja sen aiheuttamat automaattiset tyyppimuunnokset, jotka voivat johtaa väärien tyyppien tahattomaan käyttöön, ja sen myötä ohjelman odottamattomaan käyttäytymiseen. Puutteiden ja epäkohtien paikkaamiseksi on kehitetty erilaisia lisäosia ja työkaluja, joista **TypeScript** [Mic19] on viime aikoina noussut merkittävään asemaan. Stack Overflown toteuttamassa kyselytutkimuksessa [Sta19] TypeScript oli mukana 10 suosituimman ohjelmointikielen listalla ja kolmannella sijalla pidetyimpien ohjelmointikielten joukossa.

TypeScript on Microsoftin luoma ja ylläpitämä ohjelmointikieli, joka on alusta alkaen suunniteltu JavaScript-yhteensopivaksi. Tämä tarkoittaa sitä, että kaikki JavaScript-kielellä kirjoitettu ohjelmakoodi on valmiiksi yhteensopivaa TypeScriptin kanssa. TypeScript ei ole kokonaan oma kielensä vaan lisää JavaScriptiin uusia ominaisuuksia. Päällimmäinen näistä on tyyppijärjestelmä, josta kielen nimikin tulee [Tar16]. Käytännössä ohjelmakoodiin lisätään tietoa siitä, minkä tyyppisiä siinä käytetyt muuttujat ovat,

jolloin kehittäjän on helpompi pysyä perillä siitä, mitä ohjelmassa tapahtuu [Kum17]. Tyyppijärjestelmän mahdollistama staattinen koodianalyysi ja tyyppitarkistus voivat auttaa selkeyttämään ohjelman käyttäytymistä eri tilanteissa ja mahdollisesti jopa vähentämään yksikkötestien tarvetta, kun koodissa tapahtuviin automaattisiin tyyppimuunnoksiin ei tarvitse varautua yhtä kattavasti.

Tutkielmassa tutkitaan, miten hyödyllisenä tai haitallisena TypeScriptiä pidetään web-ohjelmistoteollisuuden yrityksissä työskentelevien asiantuntijoiden keskuudessa, vastaavatko asiantuntijoiden mielipiteet aikaisempia tieteellisiä tutkimustuloksia ja millä ohjelmistokehitysprosessin osa-alueilla TypeScriptin hyödyt ja haitat korostuvat. Tutkielman tutkimusmenetelmänä on kyselytutkimus, joka toteutetaan Internetissä suoritettavan kyselylomakkeen avulla.

Tutkielman rakenne on seuraava: luvussa kaksi tutustutaan yleisesti tyyppijärjestelmiin, käydään läpi miten staattinen- ja dynaaminen tyypitys erovat toisistaan, miten rakenteelliset ja nimelliset tyyppijärjestelmät eroavat toisistaan ja mitä hyötyjä erilaiset tyyppijärjestelmät voivat tarjota eri tilanteissa. Luvussa kolme käsitellään JavaScript-ohjelmointikieltä, sen historiaa, rakennetta, tyyppijärjestelmää sekä tunnistettuja puutteita, jonka jälkeen paneudutaan TypeScriptiin. TypeScriptin osalta tutustutaan kielen toimintaperiaatteeseen, tyyppijärjestelmään ja minkälaisia hyötyjä TypeScript voisi tuoda JavaScript-kielessä havaittuihin puutteisiin. Sen jälkeen esitellään aiheeseen liittyvää tutkimusta, jonka jälkeen esitellään tutkielmassa suoritettava tutkimus. Lopuksi paneudutaan tutkimuksen suorittamiseen ja esitellään tulosten perusteella tehdyt johtopäätökset sekä tutkielman yhteenveto.

2 Tyyppijärjestelmät

Nykyaikaisessa ohjelmistosuunnittelussa on käytössä laaja joukko muodollisia menetelmiä, joiden avulla voidaan varmistaa, että järjestelmä käyttäytyy oikein suhteessa sen määrittelyyn tai oletettuun toiminnallisuuteen. Yksi suosituimmista ja vakiintuneimmista näistä muodollisista menetelmistä on tyyppijärjestelmät.

Tyyppijärjestelmät on hyvin laaja käsite ja sitä on vaikea selittää sekä kattavasti, että ymmärrettävästi. Pierce kiteyttää kirjassaan *Types and Programming Languages* [Pie02, s. 1] tyyppijärjestelmien määritelmän seuraavalla tavalla:

> *"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."*

Hänen mukaansa tyyppijärjestelmät ovat pohjimmiltaan keino estää tietynlaisten virheiden, kuten suoritusaikaisten tyyppivirheiden (engl. *runtime type errors*), esiintyminen ohjelmassa [Pie02, s. 3]. Myös Cardelli kuvailee tyyppijärjestelmiä vastaavalla tavalla toteamalla, että niiden perimmäinen tarkoitus on estää virheiden esiintyminen ohjelman suorittamisen aikana [Car04, s. 1].

Käytännössä tyyppijärjestelmien avulla voidaan määrittää tyyppi jokaiselle muuttujalle, lausekkeelle, oliolle, funktiolle tai luokalle. Näitä tyyppejä käytetään yhdessä sääntöjen kanssa, joiden tarkoituksena on paljastaa ohjelmassa mahdollisesti piilevät virheet. Nämä tarkistukset voidaan suorittaa käännösaikaisesti (staattinen koodianalyysi) tai suoritusaikaisesti (dynaaminen koodianalyysi). Tyypillisiin sääntöihin kuuluu sen varmistaminen, että muuttujaan sijoitettavan arvon tyyppi vastaa muuttujan tyyppiä, tai sen

varmistaminen, että funktiokutsuissa käytetään sen tyyppisiä argumentteja, kuin mitä funktio odottaa saavansa [Fen18, s. 84].

Tyyppijärjestelmässä käytettävät tyypit toimivat kuvaajina, joiden avulla ilmaistaan järjestelmän komponenttien väliset hyväksyttävät vuorovaikutustavat. Tyyppien perusteella havaitut virheet ovat riippuvaisia tyyppijärjestelmän säännöistä sekä tarkistuksen monimutkaisuudesta [Fen18, s. 84].

2.1 Staattinen- ja dynaaminen tyypitys

Ohjemointikielistä puhuttaessa mainitaan usein termit *staattinen tyypitys* ja *dynaaminen tyypitys*. Termeillä viitataan ohjelmointikielessä käytettyyn koodin analysointitapaan, joten olisi parempi puhua staattisesta koodianalyysistä ja dynaamisesta koodianalyysistä. Koodin analysointi tarkoittaa ohjelmakoodin tutkimista ihmisen suorittamana tai automaattisesti siihen tarkoitetun työkalun avulla [Pie02, s. 1].

Staattisella koodianalyysillä tarkoitetaan ohjelmakoodin analysointia suorittamatta sitä. Tämä tekee siitä hyödyllistä esimerkiksi ohjelmien automaattista analyysia varten. Tyypillisiä käyttökohteita ovat ohjelmavirheiden havaitseminen sekä tyylitarkistuksien tekeminen [Pen14, s. 8].

Dynaamisella koodianalyysillä tarkoitetaan puolestaan käynnissä olevan ohjelman ominaisuuksien analysointia. Kun staattisessa koodianalyysissä käydään läpi ohjelman lähdekoodia ja tarkastellaan ominaisuuksia, jotka koskevat ohjelman jokaista suorituskertaa, keskitytään dynaamisessa analyysissä suoritusaikaisiin omainaisuuksiin, jotka koskevat juuri sitä kyseistä suorituskertaa [Bal99, s. 216].

2.2 Rakenteelliset ja nimelliset tyyppijärjestelmät

Tyyppijärjestelmiä on olemassa kahdenlaisia: rakenteellisia (engl. *structural*) ja nimellisiä (engl. *nominal*). Useissa C:n kaltaisissa ohjelmointikielissä, kuten esimerkiksi Javassa, on käytössä nimellinen tyyppijärjestelmä, joka tarkoittaa, että rajapintaluokkien (engl. *interface*) nimet ovat merkityksellisiä ja esimerkiksi luokka nähdään toteuttavan rajapintaluokan ainoastaan, jos se on merkittty rajapintaluokan nimellä. Luokan on siis nimenomaisesti ilmoitettava, että se toteuttaa juuri kyseisen rajapintaluokan. Rakenteellisessa tyyppijärjestelmässä rajapintaluokkien nimet ovat merkityksettömiä, joten toteutettavan rajapintaluokan nimenomaista merkintää ei tarvita ja arvo on käypä, kunhan sen rakenne vastaa vaaditun tyypin rakennetta [Pie02, s. 252].

Nimelliset tyyppijärjestelmät on tarkoitettu estämään tahattomat tyyppivastaavuudet, kun taas rakenteellisissa tyyppijärjestelmissä vahingossa tapahtuva tyyppivastaavuus on mahdollista ja jossain määrin jopa toivottavaa. Rakenteellisissa tyyppijärjestelmissä on esimerkiksi paljon helpompaa ottaa käyttöön yhteensopivia tyyppejä ilman, että on tarpeen muuttaa olemassa olevaa koodia, ja on myös mahdollista luoda tyyppejä, joita voidaan välittää ulkoiselle koodille ilman ulkoisesta luokasta perimisen tarvetta. Yksi rakenteellisen tyyppijärjestelmän merkittävimmistä eduista on, että sen avulla voidaan välttää lukematon määrä tyyppien eksplisiittisiä nimimerkintöjä, ja on myös mahdollista luoda tyypiltään luokkia tai rajapintaluokkia vastaavia nimettömiä olioita lisäämättä niihin erillisiä tyyppimerkintöjä [Fen18, s. 87].

2.3 Tyyppijärjestelmien hyödyt

Virheiden havaitseminen

Staattisen tyyppitarkistuksen ilmeisin hyöty on, että sen avulla voidaan havaita tietynlaiset ohjelmointivirheet hyvin aikaisessa vaiheessa. Aikaisessa vaiheessa havaitut virheet voidaan korjata, ennen kuin koodi ehditään ottaa käyttöön, eikä virhe silloin ehdi aiheuttaa häiriötä ohjelman toiminnnassa. Tyyppitarkistuksessa havaitut virheet on myös usein paljon helpompi paikantaa kuin vasta suoritusaikana havaitut virheet.

Tyyppitarkistaja voi myös tietyissä tapauksissa toimia erinomaisena tukityökaluna ohjelmakoodin ylläpitoa varten. Esimerkiksi, jos kehittäjällä on tarvetta muuttaa monimutkaista tietorakennetta, ei hänen tarvitse etsiä koodista kaikkia paikkoja, joihin muutos vaikuttaa, vaan hän voi ajaa tyyppitarkistuksen, jolloin se osaa ilmoittaa kaikista paikoista, joissa tyypit eivät enää täsmää ja jotka tulee korjata [Pie02, s. 4].

Abstrahointi

Toinen tärkeä tapa, jolla tyyppijärjestelmät tukevat ohjelmointiprosessia, on kurinalaiseen ohjelmarakenteeseen pakottaminen. Tyyppijärjestelmät muodostavat erityisesti suurten ohjelmistojen rakenteessa selkärangan, jonka varaan eri moduulien väliset rajapinnat rakennetaan. Moduulin rajapintaa voidaan pitää sen tyyppinä, joka tarjoaa yhteenvedon moduulin tarjoamista palveluista — eräänlaisena osittaisena sopimuksena toteuttajien ja käyttäjien välillä. Suurten järjestelmien jäsentäminen selkeillä rajapinnoilla varustettuihin moduuleihin johtaa abstraktimpaan suunnittelutyyliin, jossa rajapinnat suunnitellaan perusteellisesti ottamatta kantaa itse lopulliseen toteutukseen. Abstraktimpi suhtautuminen rajapintoihin johtaa taas yleensä parempaan

suunnitteluun [Pie02, s. 5].

Dokumentointi

Tyyppimerkinnöistä on myös hyötyä ohjelmakoodia luettaessa. Eri puolilla koodia esiintyvät tyyppimerkinnät toimivat dokumentaationa tarjoten hyödyllisiä vinkkejä ohjelman toiminnasta. Lisäksi toisin kuin kommentteihin kirjatut kuvaukset, tyyppimerkintöihin sidottu dokumentaatio ei voi vanhentua, koska tyyppitarkistus pitää huolen, että merkinnät ovat ajan tasalla [Pie02, s. 5].

3 JavaScript

JavaScript on viime vuosina noussut yhdeksi suosituimmista ja käytetyimmistä ohjelmointikielistä [Sta19]. Tässä luvussa tutustutaan kieleen tarkemmin. Ensin esitellään lyhyesti, mikä JavaScript on, ja käydään läpi kielen historiaa ja tutustutaan kielen rakenteeseen sekä tyyppijärjestelmään. Viimeisessä aliluvussa esitellään JavaScriptiin liittyviä tunnistettuja puutteita.

3.1 Yleistä

JavaScript on korkean tason, dynaaminen, tyypittämätön ja tulkattava ohjelmointikieli, joka sopii hyvin sekä olio-ohjelmointiin että funktionaaliseen ohjelmointiin [Fla11, s. 1].

JavaScriptin avulla on mahdollista toteuttaa monimutkaisia asioita verkkosivuille. Joka kerta, kun verkkosivu tekee jotain muuta kuin pelkästään näyttää staattista tietoa, on melko varmaa, että JavaScript liittyy asiaan jollain tavalla. Tähän lukeutuu esimerkiksi sivun osan päivittyminen erillään muusta sivusta, interaktiiviset kartat, animoidut 2D-/3D-grafiikat ynnä muut. JavaScript on yksi kolmesta web-sovelluskehityksen tukipilarista HTML:n sekä CSS:n rinnalla [Moz18].

JavaScript on tulkattava komentosarjakieli, jota on mahdollista käyttää oliopohjaisesti. Se on korkean tason kieli [1] , ja syntaksiltaan se muistuttaa C-, C++- sekä Java-kieliä sisältäen samalla tavalla muodostettavia `if`-lausekkeita, `while`-silmukoita sekä samoja loogisia operaattoreita kuten `&&` sekä `||`. Edellä mainittujen kielten ja JavaScriptin yhdenkaltaisuus jää kumminkin pitkälti syntaksitasolle, sillä JavaScript ei ole staattisesti tyypitetty

[1] Korkean tason ohjelmointikielissä konekieliset käskyt on korvattu ohjelmointikielen määritelmän mukaisilla käskyillä, jotka tulkataan yhdeksi tai useammaksi konekielitason käskyksi. Kääntäjä muuttaa lähdeohjelman konekielelle. Korkean tason ohjelmointikielet eivät ole riippuvaisia käytetystä piirisarjasta, riittää kun kullekin keskusyksikölle on kirjoitettu oma kääntäjänsä.

ohjelmointikieli kuten muut edellä mainitut, vaan se on dynaamisesti tyypitetty ohjelmointikieli. Tämä tarkoittaa, että muuttujilla ei ole määrättyjä tyyppejä vaan samaan muuttujaan voi esimerkiksi tallentaa ensin numeron ja myöhemmin tekstiä [Fla11, s. 29].

3.2 Historia

Vuonna 1995 Netscape Communications Corporation palkkasi henkilön nimeltä Brendan Eich kehittämään yhtiölle ohjelmointikielen, jota voitaisiin suorittaa Netscape-selaimessa. Yhtiöllä oli nopea tarve toimivalle prototyypille, joten Eichille annettiin kymmenen päivää aikaa [Rub18].

Tarve uudelle ohjelmointikielelle syntyi sen myötä, että yhdellä Netscape Communicationsin perustajista, Marc Andreessenillä, oli visio, että verkko (engl. *web*) tarvitsi tavan tulla dynaamisemmaksi. Andreessenin mielestä animaatioiden, vuorovaikutuksen ja vastaavantyyppisten pienten automaatioiden tulisi olla osana verkon tulevaisuutta. Tarvittiin siis kevyt skriptauskieli, jonka avulla voitaisiin hallita selaimen dokumenttioliomallia (engl. *Document Object Model, DOM*) ja joka ei olisi ensisijaisesti suunnattu kokeneille ohjelmistokehittäjille vaan esimerkiksi myös suunnittelijoille. HTML oli tuolloin ollut olemassa vain hyvin lyhyen aikaa, ja verkkosivujen luonti sekä rakenne oli tarpeeksi yksinkertainen, jotta muutkin kuin ohjelmistokehittäjät pystyivät omaksumaan sen. Oli siis selvää, että minkälainen uusi skriptauskieli ikinä olisikin, olisi sen oltava ymmärrettävä myös ei-ohjelmistokehittäjien näkökulmasta [Pey17].

Netscape oli 90-luvun puolivälin paikkeilla yleisin Internet-selain. Vuoden 1995 loppupuolella myös Microsoft ymmärsi, minkälaisen suosion verkko tulisi saavuttamaan, ja yhtiö käynnisti silloin Internet Explorer -projektin siinä toivossa, että sen avulla pystyttäisiin horjuttamaan Netscapen markkinahallin-

taa. Vastatoimena Netscape alkoi tekemään yhteistyötä Java-ohjelmointikielen kehittäjän, Sun Microsystemsin kanssa, jotta he voisivat yhdessä kilpailla Microsoftia vastaan [Ast15].

JavaScriptiä kehitettiin alun perin nimellä Mocha, jonka jälkeen nimi muutettiin LiveScriptiksi. Siinä toivossa, että pystyttäisiin hyödyntämään Javan ympärillä pyörivää hehkutusta ja asettamaan JavaScript Javan kanssa yhdessä käytettäväksi kieleksi, päätettiin nimi lopulta muuttaa JavaScriptiksi.

JavaScript ei siis itsessään liity Java-kieleen millään tavalla, vaan nimen taustalla on puhtaasti kaupalliset tekijät [Ast15]. JavaScript tunnetaan myös nimellä ECMAScript, joka on JavaScript-kielen standardoitu versio.

3.3 ECMAScript

Ecma International, eli eurooppalainen tieto- ja viestintäjärjestelmien standardointiyhdistys, on standardoinut JavaScript-kielen, ja sen standardoitu versio tunnetaan nimellä ECMAScript. Tämä versio kielestä toimii samalla tavalla kaikissa standardia tukevissa sovelluksissa. Yritykset voivat käyttää tätä avointa standardia kehittääkseen omia JavaScript-toteutuksiaan. ECMAScript-standardi on tallennettu kirjallisesti ECMA-262-spesifikaatioon. ECMA-262-standardi on myös hyväksytty kansainväliseksi ISO-standardiksi (International Organization for Standardization) ISO-16262 [Moz19b].

ECMAScript-spesifikaation työstäminen aloitettiin marraskuussa 1996, ja sen ensimmäinen versio julkaistiin kesäkuussa 1997. Ensimmäinen käyttökohde kielelle oli Netscapen 2.0-selainversiossa. Kieltä on käytetty kaikissa sen jälkeisissä Netscapen selainversioissa sekä kaikissa Microsoftin selaimissa versiosta 3.0 lähtien. Huhtikuussa 1998 kieli hyväksyttiin ISO-standardiksi, ja saman vuoden kesäkuussa julkaistu kielen toinen versio täytti myös standardin vaatimukset [ECM18].

ECMAScriptin kolmas versio hyväksyttiin Ecman yleiskokouksessa joulukuussa 1999, ja se julkaistiin ISO-16262:2002-standardina heinäkuussa 2002. Kolmas versio sisälsi huomattavan määrän uusia ominaisuuksia, ja se saavutti suuren yleisön hyväksynnän. Nykyään käytännössä jokainen Internet-selain tukee kyseistä versiota. ECMAScriptin kolmas versio tunnetaan myös nimellä *ES3* [ECM18].

Neljännen ECMASCript-version kehitykseen käytettiin paljon työtä, mutta sitä ei koskaan saatu päätökseen, eikä neljännettä versiota koskaan julkaistu, vaikka osaa tehdystä työstä saatiin kuitenkin hyödynnettyä kuudennen version kehityksessä [ECM18].

ECMAScriptin viidennen version avulla pyrittiin selkiyttämään joitain kolmannen version epämääräisyyksiä, ja versio toi kieleen myös paljon uusia ominaisuuksia. Viides versio julkistiin joulukuussa 2009, ja se hyväksyttiin kansainväliseksi standardiksi ISO/IEC 16262:2011. ECMAScriptin viides versio tunnetaan myös nimellä *ES5*. ECMAScript-kielen versio 5.1 julkaistiin 2011 ja sisälsi hyvin vähäisiä korjauksia edelliseen versioon nähden [ECM18].

ECMAScriptin kuudennen version, ES6:n, ensimmäinen vedos julkaistiin vuonna 2011, mutta lopullinen painos saatiin julkaistua vasta 2015. Samassa yhteydessä päätettiin vuotuisista julkaisuista ja nimeämismallin muutoksesta. Nimen tulisi jatkossa muodostua julkaisuvuoden mukaan juoksevan numeroinnin sijasta, joten ES6 tunnetaan myös nimellä ES2015 [ECM18].

Toistaiseksi vuotuisella julkaisusyklillä on julkaistu versiot ES2016, ES2017 sekä ES2018. Kyseiset versiot on julkaistu nimissä esiintyvinä vuosina [ECM18].

3.4 JavaScriptin tyyppijärjestelmä

Tietokoneohjelmat toimivat käsittelemällä arvoja, kuten numero `3.14` tai merkkijono `"Hello World"`. Jokaisella arvolla, joka voidaan ohjelmointikielessä esittää ja jota voidaan käsitellä, on aina jokin määrätty *tyyppi*. Yksi ohjelmointikielen perustavanlaatuisista ominaispiirteistä on nimenomaan sen tukemien tyyppien joukko. Kun ohjelmassa täytyy säilyttää jokin arvo tulevaa käyttöä varten, se asetetaan muuttujaan. Muuttujan avulla arvolle voidaan määrittää symbolinen nimi, ja samalla mahdollistetaan arvoon viittaaminen muuttujan avulla [Fla11, s. 29]. Tässä aliluvussa kerrotaan tarkemmin JavaScript-kielen tyyppijärjestelmästä ja sen tukemista tyypeistä.

JavaScript-tyypit voidaan jakaa kahteen ryhmään: alkeistyypit (engl. *primitive types*) ja oliotyypit (engl. *object types*). JavaScriptin alkeistyyppeihin kuuluvat numerot, merkkijonot ja totuusarvot. JavaScriptin erikoisarvot `null` sekä `undefined` ovat alkeisarvoja, mutta ne eivät ole numeroita, merkkijonoja tai totuusarvoja, vaan kumpaakin arvoa pidetään tyypillisesti oman erikoistyyppinsä ainoana jäsenenä [Fla11, s. 29].

Mikä tahansa JavaScript-arvo, joka ei ole numero, merkkijono, totuusarvo, `null` tai `undefined`, on olio. Jokainen oliotyyppiä oleva arvo on kokoelma attribuutteja (engl. *property*), joissa jokaisella attribuutilla on nimi sekä arvo. Kyseinen arvo voi olla joko alkeisarvo, kuten esimerkiksi numero tai merkkijono, tai toinen oliotyyppiä oleva arvo. Tavallinen JavaScript-olio on järjestämätön kokoelma nimettyjä arvoja. Kieli määrittelee sen lisäksi erityisen olion, joka tunnetaan taulukkona (engl. *array*), joka edustaa numeroitujen arvojen järjestettyä kokoelmaa [Fla11, s. 29]. JavaScript määrittelee taulukon lisäksi myös toisen erityisen olion, joka tunnetaan funktiona (engl. *function*). Funktio on olio, johon on liitetty suoritettavaa koodia. Funktio on kutsuttavissa, jolloin se suorittaa kyseisen koodin ja voi palauttaa laske-

tun arvon. Funktiot sekä taulukot käyttäytyvät eri lailla verrattuna toisiin olioihin, ja kieli määrittelee oman syntaksin niiden käsittelyä varten. On huomionarvoista muistaa, että JavaScript-funktiot ovat todellisia arvoja ja että JavaScript-ohjelmat voivat käsitellä niitä tavallisina olioina. Funktioita voidaan esimerkiksi tallentaa muuttujiin ja niitä voidaan välittää toisille funktioille kutsuparametreina [Fla11, s. 30].

Funktioita, jotka on luotu käytettäviksi uusien olioiden alustamisessa, kutsutaan konstruktoreiksi. Konstruktorikutsun tunnistaa siitä, että siinä käytetään operaattoria `new`. Jokainen konstruktori määrittelee olioluokan eli joukon olioita, jotka alustetaan kyseisen konstruktorifunktion avulla. Luokkia voidaan ajatella oliotyyppien alityyppeinä. Array- ja Function-luokkien lisäksi JavaScriptin ydin määrittelee kolme muuta luokkaa: *Date*, *RegExp* sekä *Error*. Date-luokan oliot edustavat päivämääriä, RegExp-luokan oliot edustavat säännöllisiä lausekkeita (engl. *regular expression*) ja Error-luokan oliot edustavat syntaksi- sekä suoritusaikaisia virheitä, joita voi esiintyä JavaScript-ohjelmissa [Fla11, s. 30].

Kuten edellä todettiin, voidaan JavaScriptin tyypit jakaa alkeistyyppeihin ja oliotyyppeihin. Nämä tyypit voidaan myös luokitella muuttuvaisiksi (engl. *mutable*) tai muuttumattomiksi (engl. *immutable*). Muuttuvaisen tyypin arvo voi muuttua. Oliot sekä taulukot ovat tyypiltään muuttuvaisia: JavaScript-ohjelma voi muuttaa olion attribuuttien ja taulukkoon kuuluvien elementtien arvoja. Numerot, totuusarvot, `null` ja `undefined` ovat tyypiltään muuttumattomia — ei ole edes järkevää puhua esimerkiksi numeron arvon muuttamisesta. Merkkijonoja voidaan ajatella yksittäisiä merkkejä sisältävinä taulukkoina ja näin ollen voisi olettaa, että ne olisivat tyypiltään muuttuvaisia. JavaScriptissä merkkijonot ovat kuitenkin muuttumattomia: mihin tahansa merkkijonotaulukon elementtiin pääsee ohjelmallisesti kä-

siksi, mutta JavaScript ei tarjoa mitään keinoa muuttaa olemassa olevan merkkijonon sisältöä luomatta uutta merkkijonoa [Fla11, s. 30].

JavaScript muuntaa tarvittaessa itsenäisesti arvoja yhdestä tyypistä toiseen. Jos ohjelma odottaa esimerkiksi merkkijonoarvoa ja sille annetaan numeroarvo, se muuntaa numeron automaattisesti merkkijonoksi. Jos käyttää arvoa, joka ei ole totuusarvo, kun totuusarvoa odotetaan, muuntaa JavaScript arvon totuusarvoksi. Tyyppimuunnosten sääntöihin paneudutaan syvemmin luvussa 3.5. JavaScriptin muunnossäännöt vaikuttavat sen samankaltaisuusmääritelmään, ja `==` yhtäsuuruusoperaattori suorittaa tyyppimuunnoksia kyseisessä luvussa kuvatulla tavalla. JavaScript-kielessä muuttujat ovat tyypittömiä: muuttujaan on mahdollista asettaa minkä tahansa tyyppinen arvo ja samaan muuttujaan voi myöhemmin asettaa eri tyyppisen arvon [Fla11, s. 31].

3.5 Tyyppimuunnokset

JavaScript-kieli on hyvin joustava sen suhteen, minkä tyyppisiä arvoja JavaScript-ohjelma tarvitsee. Luvussa 3.4 kerrottiin kielen käyttäytymisestä totuusarvojen kohdalla: kun JavaScript-koodi odottaa saavansa totuusarvon, voi sille tarjota minkä tahansa arvon, jolloin kieli muuntaa arvon automaattisesti tarvitsemaansa muotoon. Jotkut arvot (*truthy*-arvot) tulkitaan totuusarvoa käyttävässä yhteydessä *tosi*-arvona ja toiset (*falsy*-arvot) *epätosi*-arvona. Sama pätee muihin tyyppeihin: jos JavaScript haluaa merkkijonon, se muuntaa minkä tahansa sille annetun arvon merkkijonoksi. Jos JavaScript haluaa numeron, se yrittää muuntaa sille annetun arvon numeroksi. Ellei annettua arvoa pysty merkityksellisesti muuntamaan numeroksi, muuntaa JavaScript arvon erikoisarvoksi `NaN` (engl. *Not-A-Number*), joka edustaa sitä, ettei arvo ole numero [Fla11, s. 45].

Automaattiset tyyppimuunnokset saattavat tietyissä tapauksissa toimia hieman yllättävällä ja odottamattomalla tavalla. Taulukko 3.1 sisältää yhteenvedon siitä, miten JavaScript muuntaa arvoja toisikseen. Taulukon jokainen rivi edustaa tietyn tyyppistä arvoa ja jokainen sarake taas tiettyä tyyppiä. Solujen sisällä oleva tieto edustaa lopputulosta, kun rivin arvo muunnetaan vastaamaan kyseisen sarakkeen tyyppiä. Lihavoidut tiedot edustavat muunnoksia, jotka saatetaan mieltää yllättäviksi, ja tyhjät solut edustavat, ettei tyyppimuunnos ole tarpeen, joten sitä ei suoriteta.

Arvo		Lopputulos kun arvo muutetaan tyypiksi:			
		Merkkijono	**Numero**	**Totuusarvo**	**Olio**
`undefined`		`"undefined"`	`NaN`	`false`	*`TypeError`* -poikkeus
`null`		`"null"`	**`0`**	`false`	*`TypeError`* -poikkeus
`true`		`"true"`	**`1`**		`new Boolean(true)`
`false`		`"false"`	**`0`**		`new Boolean(false)`
`""`	(tyhjä merkkijono)		**`0`**	**`false`**	`new String("")`
`"1.2"`	(ei-tyhjä, numeerinen)		`1.2`	`true`	`new String("1.2")`
`"one"`	(ei-tyhjä, epänumeerinen)		`NaN`	`true`	`new String("one")`
`0`		`"0"`		**`false`**	`new Number(0)`
`-0`		`"0"`		**`false`**	`new Number(-0)`
`NaN`		`"NaN"`		**`false`**	`new Number(NaN)`
`Infinity`		`"Infinity"`		`true`	`new Number(Infinity)`
`-Infinity`		`"-Infinity"`		`true`	`new Number(-Infinity)`
`1`	(rajallinen, ei nolla)	`"1"`		`true`	`new Number(1)`
`{}`	(mikä tahansa olio)	ensin kutsutaan olion toString()-metodia ja yritetään muuntaa sen palauttama alkeisarvo merkkijonoksi, ellei toString()-metodia löydy, kutsutaan valueOf()-metodia ja yritetään muuntaa sen palauttama alkeisarvo merkkijonoksi, ellei sitäkään löydy, johtaa tilanne *TypeError* -poikkeukseen	ensin kutsutaan olion valueOf()-metodia ja yritetään muuntaa sen palauttama alkeisarvo numeroksi, ellei valueOf()-metodia löydy, kutsutaan toString()-metodia ja yritetään muuntaa sen palauttama alkeisarvo numeroksi, ellei sitäkään löydy, johtaa tilanne *TypeError* -poikkeukseen	`true`	
`[]`	(tyhjä taulukko)	`""`	**`0`**	`true`	
`[9]`	(yksi numeerinen elementti)	`"9"`	**`9`**	`true`	
`['a']`	(mikä tahansa muu taulukko)	taulukon jokainen arvo muunnetaan merkkijonoksi ja arvot yhdistetään toisiinsa pilkulla eroteltuina	`NaN`	`true`	
`function(){}`	(mikä tahansa funktio)	ensin kutsutaan olion toString()-metodia ja yritetään muuntaa sen palauttama alkeisarvo merkkijonoksi, ellei toString()-metodia löydy, kutsutaan valueOf()-metodia ja yritetään muuntaa sen palauttama alkeisarvo merkkijonoksi, ellei sitäkään löydy, johtaa tilanne *TypeError* -poikkeukseen	`NaN`	`true`	

Taulukko 3.1: JavaScriptin tyyppimuunnokset [Fla11, s. 46]

Alkeisarvon tyyppimuuntaminen toiseksi alkeisarvoksi on suoraviivaista, ja eri vaihtelut merkkijonojen ja totuusarvojen muunnoksista on esitelty melko kattavasti taulukossa 3.1. Kun jokin arvo tyyppimuunnetaan numeroksi, on syytä huomioida muutama lisäseikka. Merkkijonot, jotka sisältävät ainoastaan numeroita, muunnetaan kyseisiksi numeroiksi. Välilyönnit ovat sallittuja merkkijonon alussa tai lopussa, ja ne poistetaan tyyppimuunnosvaiheessa, mutta mitkä tahansa muut merkit, jotka eivät ole osana merkkijonolukua, aiheuttavat tyyppimuunnoksen tulokseksi arvon `NaN`. Totuusarvo `true` muuntautuu arvoksi `1`, kun taas totuusarvo `false` sekä tyhjä merkkijono `""` muuntautuvat arvoksi `0` [Fla11, s. 46].

Alkeistyypin tyyppimuuntaminen oliotyypiksi on myös melko suoraviivaista: alkeisarvot käännetään niiden kääreluokiksi, vastaavalla tavalla kuin kutsumalla konstruktoreita `String()`, `Number()` tai `Boolean()` välittämällä alkuperäinen arvo kutsuparametrina. Poikkeuksena ovat arvot `null` ja `undefined`; jos näitä arvoja käytetään ohjelman odottaessa saavansa olion, aiheuttaa se *TypeError*-poikkeuksen ohjelmassa [Fla11, s. 47].

Olioarvon tyyppimuuntaminen alkeistyypiksi on hieman monimutkaisempaa, kunhan poisluetaan tyyppimuunnos oliosta totuusarvoksi, jolloin tuloksena on aina `true` eli tosi. Muissa tapauksissa hyödynnetään olioiden metodeja `toString()` sekä `valueOf()`. Missä järjestyksessä metodeja yritetään kutsua ja miten niiden palauttamia arvoja tulkitaan kunkin alkeisarvotyypin tyyppimuunnoksessa, on kuvattu taulukossa 3.1 [Fla11, s. 49].

On syytä muistaa, että taulukot sekä funktiot ovat JavaScriptissä myös olioarvoja, joten niiden tyyppimuunnokset perustuvat samoihin sääntöihin kuin muidenkin olioarvojen, mutta lopputulos saattaa riippua esimerkiksi siitä, montako elementtiä taulukossa on ja minkä tyyppisiä elementtien arvot ovat. Taulukko 3.1 sisältää esimerkkejä siitä, millä tavalla taulukkoarvon

rakenne vaikuttaa tyyppimuunnokseen ja millä tavalla funktiot muunnetaan alkeistyypeiksi.

Koska JavaScript osaa muuntaa arvoja joustavasti, on sen löyhä yhtäsuuruusoperaattori (==) myös joustava tulkitessaan arvojen samankaltaisuuksista. Kaikki alla olevat vertailut ovat esimerkiksi totta:

```
null == undefined
"0" == 0
0 == false
false == ""
```

Löyhä yhtäsuuruusoperaattori kohtelee arvoja `null` sekä `undefined` yhdenkaltaisina, ja vertaillessaan merkkijonoja sekä totuusarvoja keskenään, se muuntaa arvot numeroiksi ennen vertailun suorittamista. Kielessä on myös toinen yhtäsuuruusoperaattori, `===`, joka käyttäytyy tiukemmin. Tiukka yhtäsuuruusoperaattori osaa esimerkiksi ottaa huomioon arvojen tyyppieroavaisuudet, eikä se löyhän yhtäsuuruusoperaattorin tavoin suorita tyyppimuunnoksia arvoille ennen vertailua.

On tärkeää pitää mielessä, että vaikka arvo muuntautuisi toiseksi, ei se tarkoita, että arvot vastaisivat toisiaan samankaltaisuusvertailussa. Esimerkiksi jos arvoa `undefined` käytetään paikassa, jossa ohjelma olettaa saavansa totuusarvon, kyseinen arvo muunnetaan arvoksi `false` eli epätosi, mutta se ei tarkoita, että vertailu `undefined == false` olisi totta.

JavaScript-lauseet ja -operaattorit odottavat saavansa tietyn tyyppisiä arvoja, ja ne suorittavat tyyppimuunnoksia saamilleen arvoille, jos tyypit eivät vastaa odotettua. Kielen `if`-lause muuntaa `undefined`-arvon `false`-arvoksi, mutta löyhä yhtäsuuruusoperaattori ei koskaan yritä muuntaa operandejaan totuusarvoiksi [Fla11, s. 47].

3.6 JavaScriptin puutteet

JavaScript-kielessä katsotaan olevan joitakin puutteita tai huonosti suunniteltuja ominaisuuksia, joista on hyvä olla tietoinen ja joita kannattaa mahdollisesti välttää [Cro08, s. 1]. Myös Fenton [Fen18, s. xxiii] kirjoittaa erilaisista JavaScript-kielessä piilevistä ansoista, joihin lähes jokainen JavaScript-kehittäjä on jossain vaiheessa törmännyt. Tässä aliluvussa esitellään tunnistetut ongelmakohdat ja paneudutaan niihin tarkemmin.

Samankaltaisuusvertailu

Cockford [Cro08, s. 109] mainitsee yhtenä JavaScriptin huonona ominaisuutena kielen samankaltaisuusvertailua. Kuten jo luvun 3.5 lopussa esiteltiin, on samankaltaisuusvertailuja mahdollista suorittaa käyttäen joko löyhää yhtäsuuruusoperaattoria `==` tai tiukkaa yhtäsuuruusoperaattoria `===` ja löyhän yhtäsuuruusoperaattorin ennen vertailua suorittamat tyyppimuunnokset voivat usein johtaa virheisiin. Yleisesti pidetään hyvänä käytäntönä käyttää joka tilanteessa tiukkaa samankaltaisuusvertailua.

Automaattinen puolipisteen sijoitus

JavaScriptissä jotkut lauseet vaativat puolipisteen lauseen päättymisen merkiksi, ja sen vuoksi kielessä on olemassa ominaisuus, jossa puolipiste sijoitetaan automaattisesti tarvittuihin kohtiin, mikäli siellä ei sellaista jo ole (engl. *automatic semicolon insertion*). Alla on kuvattuna lauseet, joita automaattinen puolipisteen sijoitus koskee [Moz19c]:

- Tyhjä lause (pelkkä `;`)
- Muuttujan määrittely (`const`, `let`, `var`)
- `import`, `export`

- Sijoituslause
- `debugger`
- `continue`, `break`, `throw`
- `return`

Vaikka ominaisuus toimii suurimman osan ajasta kehittäjälle näkymättömällä tavalla, saattaa ominaisuus tietyissä tilanteissa johtaa ohjelman toimimiseen odottamattomalla tavalla. Koodiesimerkissä 3.1 on esimerkki tällaisesta tilanteesta. Koodiesimerkissä kehittäjä on yrittänyt luoda tilanteen, jossa `return`-lauseen avulla palautetaan olio, joka sisältää avain-arvoparin `{foo: 'bar'}`. Koska aaltosulku on laitettu seuraavalle riville, sijoittaa puolipisteen automaattinen sijoitus puolipisteen `return`-lauseen perään ja palautusarvona onkin `undefined` eli määrittelemätön. Esimerkin lopussa on havainnollistettu, miten ongelmalta voitaisiin välttyä sijoittamalla aaltosulku samalle riville `return`-lauseen kanssa ja näin ollen saada palautettua toivottu arvo [Cro08, s. 102].

```
1  // palauttaa arvon undefined
2  return
3  {
4      foo: 'bar'
5  };
6
7  // palauttaa olion
8  return {
9      foo: 'bar'
10 };
```

Koodiesimerkki 3.1: Automaattinen puolipisteen sijoitus [Cro08, s. 102]

Prototyyppipohjainen periytyminen

JavaScript saattaa olla hieman sekava kehittäjille, joilla on kokemusta luokkapohjaisista kielistä, kuten *Java* tai *C++*, koska se on dynaaminen eikä sinänsä tunne luokkakäsitettä. Luokka-avainsana `class` esiteltiin ECMAScript versiossa ES2015, mutta se on vain syntaktinen harha, sillä taustalla JavaScript on edelleen prototyyppipohjainen ohjelmointikieli [Moz19a].

Prototyyppipohjainen ohjelmointi on olio-ohjelmoinnin tyyli, jota esiintyy pääasiassa tulkituissa dynaamisissa kielissä. JavaScriptissä prototyyppipohjainen periytyminen tarkoittaa sitä, että perinnöllisyyteen liittyen kielessä on vain yksi rakennelma: oliot. Kullakin oliolla on yksityinen attribuutti, joka sisältää viittauksen toiseen olioon, joka tunnetaan alkuperäisen olion prototyyppinä. Viitatulla prototyyppioliolla on oma prototyyppinsä ja ketju jatkuu eteenpäin, kunnes jollain oliolla on prototyyppinä arvo `null`. Määritelmän mukaan `null`-arvolla ei ole prototyyppiä, ja se toimii prototyyppiketjun lopullisena linkkinä [Moz19a].

Käytännössä prototyyppipohjainen periytyminen toimii sillä tavalla, että jos oliolle kutsutaan metodia, jota siltä itseltään ei löydy, sitä etsitään seuraavaksi olion prototyypistä. Jos metodia ei löydy prototyypistäkään, jatketaan etsintää prototyyppiketjussa ylöspäin, kunnes se löytyy. Jos metodia ei löydy lainkaan prototyyppiketjusta tai oliolta itseltään, palautuu `undefined` [Fla11, s. 122].

Vaikka prototyyppipohjaisessa periytymisessä ei itsessään ole vikaa, pidetään sitä usein vaikeatajuisena ja sen myötä yhtenä JavaScriptin heikkouksista [Moz19a].

Automaattiset tyyppimuunnokset

JavaScript on aina tukenut dynaamisesti tyypitettyjä muuttujia, minkä seurauksena se joutuu tekemään suoritusaikaisia ponnisteluja päätelläkseen tyypityksiä ja suorittaakseen tyyppimuunnoksia saadakseen sellaiset lausekkeet toimimaan, jotka aiheuttaisivat staattisesti tyypitetyssä kielessä virheen. Luvussa 3.5 esiteltiin, miten tyyppimuunnokset toimivat, ja yleisimmät muunnokset koskevat sitä, että muuttujan arvo on tarve muuntaa joko merkkijonoksi, numeroksi tai totuusarvoksi. Aina kun arvo yhdistetään merkkijonoon, muunnetaan kyseinen arvo merkkijonoksi; aina kun suoritetaan matemaattinen operaatio, yritetään arvo muuntaa numeroksi; ja aina kun arvoa käytetään loogisessa operaatiossa, muunnetaan arvo ensin totuusarvoksi.

Joissakin tapauksissa automaattinen tyyppimuunos voi olla hyödyllinen ominaisuus, erityisesti lyhyiden loogisten lausekkeiden luomisessa. Muissa tapauksissa automaattinen tyyppimuunnos saattaa piilottaa väärien tyyppien tahattoman käytön ja johtaa ohjelman odottamattomaan käyttäytymiseen [Fen18, s. xxiii].

Erään tutkimuksen [OBPM13] mukaan jopa 74 % JavaScript-virheistä liittyvät siihen, että funktioita kutsutaan odottamattomalla tai virheellisellä parametrilla tai että olion attribuutiksi yritetään asettaa odottamatonta tai virheellistä arvoa. Esimerkiksi, jos JavaScriptin `Date`-olion `setDate()`-metodia kutsutaan merkkijonoarvolla, kun metodi odottaa sitä kutsuttavan kokonaisluvulla.

Koodiesimerkissä 3.2 voi nähdä esimerkin, miten automaattinen tyyppimuunnos toimii käytännössä ja miksi se saattaa johtaa odottamattomiin tuloksiin JavaScript-ohjelmassa. Esimerkissä numeroarvo `1` yhdistetään ensin plusmerkin avulla merkkijonoarvoon `'0'`, jolloin lopputulokseksi saadaan merkkijonoarvo `'10'`. Esimerkin lopussa kyseinen arvo kerrotaan kahdella,

jolloin merkkijonoarvo muuttuu numeroarvoksi 20.

```
const num = 1;
const str = '0';

// tuloksena on '10', ei 1
const strTen = num + str;

// tuloksena on 20
const result = strTen * 2;
```

Koodiesimerkki 3.2: Automaattinen tyyppimuunnos JavaScript-koodissa [Fen18, s. xxiv]

Näkyvyysalueet

Useimmissa moderneissa C-pohjaisissa ohjelmiointikielissä aaltosulkeet `{}` luovat uuden näkyvyysalueen. Kun muuttuja alustetaan aaltosulkeiden välissä, ei muuttuja ole lainkaan käytettävissä sulkeiden luoman näkyvyysalueen ulkopuolella [Cro08, s. 102].

Perinteisesti JavaScript on kumminkin toiminut eri tavalla; käyttämällä funktioihin sidottuja näkyvyysalueita, mikä tarkoittaa, että aaltosulkeilla eristetyillä koodilohkoilla ei ole vaikutusta näkyvyysalueisiin. Sen sijaan muuttujat kuuluvat sen funktion näkyvyysalueeseen, jonka sisällä ne on alustettu, tai globaaliin näkyvyysalueeseen (engl. *global scope*), jos muuttujia ei ole alustettu funktion sisällä. Flanaganin [Fla11, s. 269] mukaan lohkoihin sidottujen näkvyvyysalueiden puuttumista on kauan pidetty JavaScript-kielen puutteena. ES6 sisälsi avainsanat `let` sekä `const`, joiden avulla voidaan nyttemmin alustaa muuttujia, joiden näkyvyysalue on sidottu lohkoihin, mutta valitettavasti on olemassa vielä suuri määrä ohjelmistoja, joissa on käytössä

sitä vanhempi versio kielestä ja joissa ei vielä tueta uusia muuttujatyyppejä.

Myös `var`-avainsanan tahaton poisjättäminen alustettaessa muuttujaa funktion sisällä voi johtaa ongelmiin, koska silloin muuttuja ylennetään globaaliin näkyvyysalueeseen ja sen arvoa on mahdollista muuttaa funktion ulkopuolella. Muuttujien nostaminen (engl. *hoisting*), jonka myötä kaikki muuttujat käyttäytyvät kuin ne olisi alustettu heti funktion alussa, saattaa myös johtaa komplikaatioihin, jos kehittäjä ei tiedä kielen käyttäytyvän kyseisellä tavalla [Cro08, s. 113].

Mainituista näkyvyysalueisiin liittyvistä hankalista yllätyksistä huolimatta tarjoaa JavaScript hyvin tehokkaan mekanismin, jonka perusideana on, että funktion parametrina annettu funktio kietoo mukaansa sitä ympäröivän näkyvyysalueen. Kun funktio myöhemmin suoritetaan, käytetään silloin määrittelyn aikaista nimiavaruutta. Tätä mekanismia kutsutaan sulkeumaksi (engl *closure*). Sulkeumaan suljettuja muuttujia voidaan muokata sulkeuman sisältä, vaikka ne eivät olisi sulkeuman näkyvyysalueella, ja Fentonin mukaan sulkeumat ovat yksi JavaScriptin tehokkaimmista ominaisuuksista [Fen18, s. xxv].

Koodiesimerkissä 3.3 `createCounter`-funktio määrittelee muuttujan `x`, sulkee sen sulkeumaan `function(){x++; return x;}` ja palauttaa tuon sulkeuman arvonaan. Jokainen `createCounter`-funktion kutsu luo uuden sulkeuman ja jokaisella sulkeumalla on oma `x`-muuttujansa, jonka lähtöarvona on nolla. Jokainen sulkeumakutsu puolestaan lisää yhden kyseisen sulkeuman `x`-arvoon ja palauttaa muuttujan uuden arvon. Sulkeuman ulkopuolella muuttuja `x` pysyy täysin piilossa, eikä siihen pääse käsiksi.

```
function createCounter() {
  var x=0;
  return function() {x++; return x;} // palautusarvo on funktio!
}

var counter1 = createCounter();
write("counter 1 = " + counter1()); // 1
write("counter 1 = " + counter1()); // 2
write("counter 1 = " + counter1()); // 3

var counter2 = createCounter();
write("counter 2 = " + counter2()); // 1 !
write("counter 2 = " + counter2()); // 2
write("counter 1 = " + counter1()); // 4 !

var counter3 = createCounter();
write("counter 3 = " + counter3()); // 1
```

Koodiesimerkki 3.3: Sulkeuma JavaScript-koodissa

Dynaaminen tyypitys

Tietokoneohjelmassa *muuttuja* on muistipaikka, jossa ohjelma voi säilyttää tarvitsemaansa tietoa. Muuttujan *nimi* toimiii tunnuksena, jolla siihen viitataan ohjelman koodissa. Muuttujan *arvo* taas kuvastaa muuttujan sisältöä tietyllä ohjelman suorituksen hetkellä.

Staattisesti tyypitetyssä kielessä, kuten esimerkiksi Javassa, muuttujan tyyppi on annettava jo esittelyvaiheessa, eikä muuttujaan ole sen jälkeen mahdollista asettaa sellaista arvoa, jonka tyyppi ei vastaa muuttujan tyyppiä. JavaScript-kielessä on puolestaan dynaaminen tyypitys, joka tarkoittaa, että itse muuttujilla ei ole tyyppiä, vaan arvoilla on tyyppi. Muuttujaan voidaan

esimerkiksi sen alustuksen yhteydessä asettaa merkkijono, mutta myöhemmin samaan muuttujaan voidaan asettaa esimerkiksi numero, olio tai jopa funktio.

Dynaamisen tyypityksen ansiosta muuttujan arvon tyyppi on tiedossa vasta suoritusaikaisesti, jolloin ohjelmassa saattaa esiintyä virheitä, jos kehittäjä ei ole osannut varautua eri tyyppisiin arvoihin. Dynaamisen tyypityksen ansiosta kehitystyökaluilla ei myöskään ole käytettävissään kuin paras arvaus muuttujien arvojen tyypistä, joten ne eivät useinkaan pysty tarjoamaan kehittäjille hyödyllisiä aputyökaluja, kuten automaattista koodintäydennystä tai tyyppivihjeitä, jotka eivät ole liian yleisluontoisia ollakseen hyödyllisiä [Fen18, s. xxv].

4 TypeScript

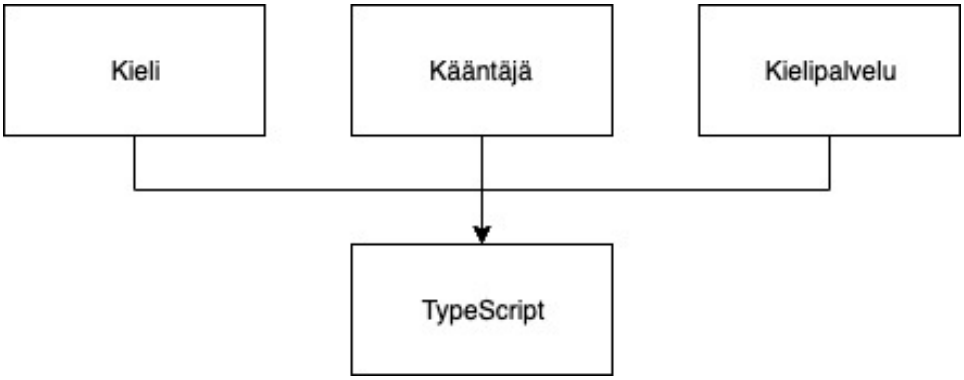
TypeScript on Microsoftin luoma ja ylläpitämä kieli, joka on julkaistu vapaan ohjelmiston Apache 2.0 -lisenssillä (2004). Kieli keskittyy mahdollistamaan useista tuhansista koodiriveistä kirjoitettujen JavaScript-ohjelmistojen kehittämisen. Itse asiassa Microsoft on kirjoittanut sekä Azure Management Portal -ohjelmiston (1,2 milj. koodiriviä) että Visual Studio Code -ohjelmointiympäristön (300 000 koodiriviä) TypeScriptin avulla. Kieli tarjoaa laajamittaisten JavaScript-ohjelmistojen kehittämisen tueksi parempia suunnittelunaikaisia työkaluja, käännösaikaisen koodianalyysin sekä dynaamisten moduulien suoritusaikaisen lataamisen [Fen18, s. xix].

Tässä luvussa tutustutaan tarkemmin TypeScript-kieleen, siihen mikä sen toimintaperiaate on ja minkälaiset ovat sen perusominaisuudet. Luvussa esitellään myös TypeScriptin mahdollisia hyötyjä ja lopuksi esitellään mahdollisia vaihtoehtoja TypeScriptille.

4.1 Toimintaperiaate

TypeScript on JavaScript-kielen laajennus (engl. *superset*), joka lopulta käännetään tavalliseksi JavaScript-koodiksi, jonka version (ECMAScript 3 tai uudempi) kehittäjä pystyy itse päättämään [Fen18, s. 1]. Se, että TypeScript on JavaScript-kielen laajennus tarkoittaa, että TypeScript sisältää kaikki JavaScriptin ominaisuudet, mutta tarjoaa siihen lisäksi myös muita ominaisuuksia. TypeScriptin tarjoamia lisäominaisuuksia ei ole pakko käyttää, vaan itse asiassa kaikki olemassa oleva JavaScript-koodi on myös käypää TypeScript-koodia [Tar16].

TypeScript koostuu kolmesta erillisestä, mutta toisiaan täydentävästä osasta, jotka on esitetty kuvassa 4.1.



Kuva 4.1: TypeScriptin eri osat

Kieli koostuu JavaScriptiin verrattuna hieman erilaisesta syntaksista, avainsanoista sekä tyyppimerkinnöistä (engl. *type annotations*). Listatuista kolmesta osasta kieli on se, johon kehittäjä on eniten suorassa kosketuksessa.

Kääntäjä suorittaa tyyppitietojen poistamisen ja koodimuunnokset, joiden avulla TypeScript-koodi muunnetaan JavaScript-koodiksi. TypeScript ei siis tuota aidosti staattisesti tyypitettyä ohjelmakoodia, vaan kaikki tyyppeihin viittaava poistetaan kääntämisen yhteydessä [Tar16]. Perinteisesti, kun puhutaan ohjelmointikielen kääntämisestä, tarkoitetaan muunnosta, jossa ihmiselle helpossa muodossa oleva lähdekoodi muutetaan tietokoneen ymmärtämään muotoon. TypeScriptin tapauksessa lähdekoodi käännetään toiseen lähdekoodiin. Kääntäjä osaa lisäksi ilmoittaa havaitessaan aihetta varoitukseen tai virheen koodista, ja sen voi tarvittaessa määrätä suorittamaan lisätehtäviä, kuten esimerkiksi tuloksena syntyvän JavaScript-koodin yhdistämisen yhteen ainoaan tiedostoon [Fen18, s. xxii].

Kielipalvelu (engl. *language service*) tarjoaa ohjelman koodista kerättyjen tyyppitietojen perusteella tyyppitietoa sellaisessa muodossa, josta kehitystyökalut voivat sitä käyttää tarjotakseen esimerkiksi koodin automaattitäydennystä, tyyppivihjeitä sekä mahdollisia refaktorointivaihtoehtoja [Fen18, s. xxii].

4.2 TypeScriptin tyyppijärjestelmä

TypeScriptissä on rakenteellinen tyyppijärjestelmä, jossa hyödynnetään käännösaikaista staattista tyyppitarkistusta. Tässä aliluvussa esitellään tyyppijärjestelmän erityispiirteitä, kuten valinnaisia staattisia tyyppejä ja tyyppitiedon poistoa. Lisäksi käydään läpi, miten tyyppipäättely ja tyyppitarkistus toimii TypeScriptissä ja miten on mahdollista määrittää tyyppejä myös ympäröivälle koodille, joka ei välttämättä ole edes TypeScriptillä kirjoitettua.

Valinnaiset staattiset tyypit

Kuten jo aikaisemmin nostettiin esille, on JavaScript dynaamisesti tyypitetty kieli; muuttujilla ei siis ole kiinteää tyyppiä, joten operaatioihin ei voida soveltaa minkäänlaisia rajoituksia. Yhdentyyppinen arvo voidaan sitoa muuttujaan, ja täysin erityyppinen arvo voidaan myöhemmin sitoa samaan muuttujaan. Operaatioita pystyy suorittamaan yhteensopimattomilla arvoilla, ja sen myötä voidaan saada myös arvaamattomia tuloksia. Funktioita kutsuttaessa ei ole olemassa mitään mekanismia, joka pakottaisi käyttämään ainoastaan oikeantyyppisiä argumentteja, ja funktiokutsuissa on jopa mahdollista käyttää liian montaa tai liian vähän argumentteja. Tämän ansiosta JavaScriptin tyyppijärjestelmä on erittäin joustava, mutta kuten luvussa 3.6 tuli esille, voi samainen joustavuus myös aiheuttaa ongelmia.

TypeScript tarjoaa järjestelmän tyyppien päättelyä ja määrittelyä varten, mutta mahdollistaa tyyppien valinnaisuuden. Valinnaisuus on tärkeää, koska se tarkoittaa, että kehittäjä pystyy itse valitsemaan, milloin pakottaa tietyt tyypit ja milloin sallia dynaamiset tyypit. Ellei tyyppitarkistusta ohiteta käyttämällä `any`-tyyppiä, yrittää kääntäjä rajata ohjelmassa esiintyvät tyypit tarkistamalla sekä päätellyt tyypit että tyyppimerkinnöillä eksplisiittisesti merkityt tyypit. Kaikki tarkistukset tehdään käännösaikaisesti, mikä tekee

TypeScriptistä staattisesti tyypitetyn kielen. On kääntäjän vastuulla laatia luettelo käytetyistä tyypeistä, tarkistaa lausekkeet näitä tyyppejä vastaan ja poistaa kaikki tyyppitiedot, kun käännetään TypeScript-koodi JavaScript-koodiksi [Fen18, s. 85].

Tyyppitiedon poisto

Käännettäessä TypeScript-koodia JavaScript-koodiksi eroaa generoitu koodi alkuperäisestä koodista kahdesta syystä: koodimuunnoksen (engl. *code transformation*) ja tyyppitiedon poiston (engl. *type erasure*) ansiosta. Koodimuunnos muuntaa kieliominaisuudet, jotka eivät ole käytettävissä kohde-JavaScript-versiossa, toimiviksi vaihtoehdoiksi kyseisille ominaisuuksille. Esimerkiksi jos kohdeversiona on ECMAScript 5, jossa luokat eivät ole vielä mahdollistettuja, kaikki TypeScriptissä luodut luokat muunnetaan välittömästi suoritettaviin funktiolausekkeisiin (engl. *immediately invoked function expression*), joiden avulla voidaan siinä versiossa saavuttaa vastaava toiminnallisuus. Tyyppitiedon poisto on prosessi, jossa tyyppimerkinnät poistetaan lähdekoodista, koska JavaScript-tulkki ei niitä ymmärrä.

Tyyppitiedon poistossa koodista poistetaan tyyppimerkinnät, itse luodut tyypit sekä rajapintaluokat (engl. *interface*). Edellä mainitut ovat tarpeellisia vain kehitys- ja käännösvaiheessa staattista koodianalyysiä varten. Tyyppejä ei käytetä suoritusaikaisesti, joten tyyppitiedot ovat silloin tarpeettomia. Suoritusaikaisia tyyppeihin liittyviä ongelmia ei siitä huolimatta tulisi ilmetä, sillä tyyppien looginen käyttö koodissa on tarkistettu jo käännösvaiheessa. Poikkeuksina ovat tilanteet, joissa käytetään rajapintojen kautta tulevia ulkoisia tarkistamattomia syötteitä tai dynaamisia tyyppejä mahdollistavaa `any`-tyyppiä.

Generoitu JavaScript-koodi muistuttaa tyyppitiedon poistosta ja koodi-

muunnoksesta huolimatta yhä hyvin paljon alkuperäistä TypeScript-koodia. Kaikki alkuperäinen toiminnallisuus säilytetään, ja lopulliset erot riippuvat paljon siitä, mitä lähde- ja kohdeversiota JavaScriptistä käytetään [Fen18, s. 88-89].

Tyyppipäättely

Tyyppipäättelyä (engl. *type inference*) voisi ajatella tyyppitiedon poiston suorana vastakohtana. Tyyppipäättelyllä tarkoitetaan prosessia, jossa oikea tyyppi päätellään koodin perusteella eksplisiittisten tyyppimerkintöjen puuttuessa joko osittain tai kokonaan.

Yksinkertaisimmillaan tyyppitiedon päättely voi toimia sillä tavalla, että kun uuteen muuttujaan asetetaan arvo ja kyseisen arvon tyyppi on jo tiedossa, päättelee TypeScript uuden muuttujan tyypiksi saman tyypin, kuin mitä siihen asetetulla arvolla on. Tällaisen, melko suoraviivaisen päättelyn lisäksi, osaa TypeScript myös suorittaa paljon monimutkaisempia päättelyitä ja ottaa ne huomioon tyyppitarkistuksessa. TypeScript suorittaa syvällisiä tarkistuksia, joiden pohjalta se muodostaa luettelon ohjelmassa käytetyistä tyypeistä ja vertailee sen jälkeen määrittelyjä, lausekkeita ja operaatioita luetteloa vasten. Kuvatun prosessin aikana, jos tyyppitietoa ei ole saatavilla, osaa TypeScript hyödyntää erilaisia tekniikoita päätelläkseen oikean tyypin.

Koodiesimerkki 4.1 osoittaa, kuinka tyyppipäättelyä voidaan suorittaa asteittain epäsuoremmalla tavalla. Funktion `add` palautusarvo päätellään askeltamalla taaksepäin funktion palautusarvolauseesta. Palautusarvon tyyppi päätellään arvioimalla ilmaisun `a + b` tyyppi, joka puolestaan päätellään tarkastelemalla yksittäisten parametrien tyyppejä.

Koodiesimerkin 4.1 viimeisessä lausekkeessa alustetun nimettömän funktion `result`-parametrin tyyppi voidaan päätellä sen perusteella, missä asiayh-

teydessä funktio luodaan. Koska nimetöntä funktiota käytetään `callsFunction`-funktion kutsussa, ja `callsFunction` toteuttaa `CallsFunction`-rajapintaluokan, osaa kääntäjä tyyppitiedoista päätellä, että `result`-parametri tulee olemaan tyypiltään merkkijono [Fen18, s. 90].

```
function add(a: number, b: number) {
    /* The return value is used to determine
       the return type of the function */
    return a + b;
}

interface CallsFunction {
    (cb: (result: string) => any): void;
}

// The cb parameter is inferred to be a function accepting a
    string
const callsFunction: CallsFunction = function (cb) {
    cb('Done');

    // Error: Argument of type '1' is not assignable to
    parameter of type string
    cb(1);
}

// The result parameter is inferred to be a string
callsFunction(function (result) {
    return result;
});
```

Koodiesimerkki 4.1: Tyyppipäättelyä TypeScriptissä [Fen18, s. 90]

Tyyppitietoja pääteltäessä saattaa tulla eteen tilanteita, joissa on määriteltävä paras yhteinen tyyppi (engl. *best common type*). Koodiesimerkki 4.2 osoittaa tilanteen, jossa taulukkoliteraalin kaikkien elementtien arvot on huomioitava, jotta voidaan päätellä paras yhteinen tyyppi taulukon kaikille arvoille.

Esimerkissä luodaan ensin luokka `Shape`, jonka jälkeen luodaan kaksi muuta luokkaa `Square` sekä `Circle`, jotka molemmat ovat laajennuksia luokasta `Shape`. Muuttujan `xs1` tyypiksi päätellään taulukko, joka sisältää `Shape`-tyyppisiä olioita (`Shape[]`), koska kaikki sen sisältämät arvot ovat joko tyyppiä `Shape` tai siitä perittyjä arvoja. Mikä saattaa yllättää on, että muuttujan `xs2` tyypiksi ei päätellä `Shape[]`, vaikka kaikki sen sisältämät arvot ovat perittyjä `Shape`-tyypistä, vaan sen tyypiksi päätellään taulukko, joka voi sisältää joko `Circle`- tai `Square`-tyyppisiä olioita. Syynä tähän on, että taulukko ei sisällä yhtään arvoa, joka on suoraan `Shape`-tyyppinen. Esimerkin lopussa havainnollistetaan, miten tyyppimerkinnällä voidaan asettaa muuttujan `xs3` tyypiksi `Shape[]`, vaikka se sisältää samat arvot kuin edellisellä rivillä oleva muuttuja `xs2`.

```
1  class Shape {
2      color: string;
3  }
4  class Square extends Shape {
5      sideLength: number;
6  }
7  class Circle extends Shape {
8      radius: number;
9  }
10
11 var xs1 = [new Shape(), new Square(), new Circle()]; // Shape[]
12 var xs2 = [new Square(), new Circle()]; // (Square | Circle)[]
13 var xs3: Shape[] = [new Square(), new Circle()]; // Shape[]
```

Koodiesimerkki 4.2: Paras yhteinen tyyppi

Parhaan yhteisen tyypin päättelyä ei käytetä ainoastaan taulukkojen kohdalla, vaan sitä käytetään aina, kun tyyppi on pääteltävä sellaisessa

tilanteessa, jossa useammalla arvolla voi olla eri tyyppejä. Tällaisia tilanteita voisi olla esimerkiksi funktion palautusarvon päättely silloin, kun funktio sisältää useamman palautusarvolausekkeen [Fen18, s. 90].

Asiayhteydelliset tyypit (engl. *contextual types*) ovat hyvä esimerkki siitä, miten edistyneesti tyyppipäättely voi toimia. Asiayhteydellisessä tyypityksessä kääntäjä päättelee ilmaisun tyypin sen sijainnin perusteella. Koodiesimerkissä 4.3 `event`-parametrin tyyppi päätellään `window.onclick`-funktion tiedossa olevan tyyppitiedon perusteella. Päätelmä ei rajoitu pelkästään parametreihin, vaan myös palautusarvon tyyppi voidaan päätellä, koska `window.onclick`-funktion tyyppitiedot ovat kääntäjän tiedossa.

```
1 window.onclick = function(event) {
2     var button = event.button();
3 };
```

Koodiesimerkki 4.3: Asiyhteydelliset tyypit [Fen18 s. 91]

TypeScript osaa koodiesimerkin 4.3 asiayhteydestä päätellä esimerkiksi, että funktion `event`-parametrin tyyppi on `mouseEvent` ja että funktion palautustyyppi on `any`, vaikka mitään tyyppitietoja ei ole eksplisiittisesti ilmaistu koodissa [Fen18, s. 91].

Tyyppitarkistus

Kun ohjelmakoodi käännetään TypeScript-koodista JavaScript-koodiksi TypeScript-kääntäjä analysoi koodin ja kokoaa kaikista käytetyistä tyypeistä luettelon. Luetteloa hyödynnetään tyyppitarkistuksen yhteydessä varmistamaan, ettei tyyppejä käytetä väärin. Yksinkertaisimmillaan kääntäjä tarkistaa, että kun kutsutaan funktiota, joka hyväksyy `number`-tyyppisen parametrin, funktion kaikissa kutsuissa välitetään argumentti, jonka tyyppi on

yhteensopiva `number`-tyypin kanssa [Fen18, s. 93].

Koodiesimerkki 4.4 esittää sarjaa kelvollisia kutsuja funktioon, jonka parametri on nimeltään `input` ja sen tyyppi on `number`. Argumentteina sallitaan vain arvot, joiden tyyppinä on `number`, `enum`, `null`, `undefined` tai `any`. On huomionarvoista muistaa, että `any`-tyyppi sallii dynaamisen käyttäytymisen TypeScriptissä ja sitä käytettäessä jää kehittäjän vastuulle, että arvo tulee olemaan hyväksyttävä suoritusaikaisesti [Fen18, s. 93].

```
1  function acceptNumber(input: number) {
2      return input;
3  }
4
5  // number
6  acceptNumber(1);
7
8  // enum
9  acceptNumber(Size.XL);
10
11 // null
12 acceptNumber(null);
```

Koodiesimerkki 4.4: Parametrin tyyppitarkistus [Fen 18, s. 93]

Tyyppien muuttuessa monimutkaisemmiksi edellyttää tyyppitarkistus kohteiden tarkempaa tarkastelua. Oliota tarkistettaessa sen jokaista attribuuttia testataan ja niiden tyyppitietoja verrataan odotettuihin tyyppeihin. Jotta olioiden tyypit vastaisivat toisiaan, tulee olioiden julkisilla attribuuteilla olla samat nimet sekä tyypit ja olioiden julkisten metodien parametrien sekä paluuarvon tyyppien on täsmättävä. Jos olion attribuuttina on toinen olio, suoritetaan sille ja kaikille sen mahdollisille jäsenolioille vastaavanlainen tyyppitarkistus kuin edellä kuvailtiin.

Koodiesimerkissä 4.5 esitetään kolme erinimistä luokkaa sekä yksi olioliteraali, jotka kaikki ovat kääntäjän näkökulmasta samantyyppisiä. Alla olevan esimerkin merkittävimmät osat ovat `name`-attribuutti sekä `show`-metodi. Olion on sisällettävä julkinen `name`-niminen attribuutti, jonka on oltava merkkijono. Ei ole merkitystä, onko kyseinen attribuutti konstruktoriparametri tai ei. Olion on lisäksi sisällettävä julkinen `show`-niminen metodi, jonka palautusarvon tulee olla yhteensopiva `number`-tyypin kanssa. Metodin parametrien on myös oltava yhteensopivia — esimerkkitapauksessa valinnainen `hint`-parametri voidaan täsmätä käyttämällä oletusparametria tai jättämällä parametri kokonaan pois. Parametrin valinnaisuus ilmaistaan TypeScriptissä lisäämällä parametrin nimen perään kysymysmerkki (`?`). Mikäli luokalla `C1` olisi ollut pakollinen `hint`-parametri, se ei olisi ollut yhteensopiva koodiesimerkissä 4.5 esitettyjen muiden tyyppien kanssa. Kuten neljännestä tyypistä voi huomata, voivat olioliteraalit olla yhteensopivia luokkien kanssa, kunhan ne vain läpäisevät tyyppivertailun [Fen18, s. 94].

```
class C1 {
    name: string;
    show(hint?: string) {
        return 1;
    }
}

class C2 {
    constructor(public name: string) {
    }
    show(hint: string = 'default') {
        return Math.floor(Math.random() * 10);
    }
}

class C3 {
    name: string;
    show() {
        return 'Dynamic' as any;
    }
}

var T4 = {
    name: '',
    show() {
        return 1;
    }
}

var c1 = new C1();
var c2 = new C2('A name');
var c3 = new C3();

// c1, c2, c3 and T4 are equivalent
var arr: C1[] = [c1, c2, c3, T4];

for (var i = 0; i < arr.length; i++) {
    // show() is found and callable for all elements in arr
    arr[i].show();
}
```

Koodiesimerkki 4.5: Yhteensopivat tyypit [Fen18, s. 94]

Ympäröivän koodin tyyppimäärittely

Ympäröivän koodin tyyppimäärittelyä (engl. *ambient declarations*) voidaan käyttää tyyppitiedon lisäämiseksi olemassa olevaan JavaScript-koodiin. Fentonin mukaan ominaisuutta käytetään yleisesti tyyppitiedon lisäämiseksi jo olemassa olevaan itse luotuun koodiin tai kolmannen osapuolen kirjastoihin, joita halutaan käyttää TypeScript-ohjelmassa. Lisäämällä tyyppitietoa ulkoiseen, mutta ohjelmassa käytettävään koodiin, voi kääntäjä antaa varoituksia, mikäli tyyppejä käytetään väärin, ja lisättyjen tyyppitietojen avulla voidaan kehitysvaiheessa saada myös tarkempia automaattitäydennysvihjeitä [Fen18, s. 95-96].

4.3 TypeScriptin hyödyt suhteessa JavaScriptiin

Luvussa 3.6 käytiin läpi JavaScriptiin liittyviä tunnistettuja puutteita. Tässä aliluvussa tarkastellaan TypeScriptin hyötyjä käymällä läpi, millä tavalla se voi tarjota ratkaisuja aiemmin esitettyihin ongelmakohtiin.

Samankaltaisuusvertailu

Jos yritetään suorittaa samankaltaisuusvertailuja, joko löyhiä tai tiukkoja, sellaisten arvojen välillä, joiden tyypit eivät täsmää, ilmoittaa TypeScriptin staattinen koodianalyysi virheestä. Näin ollen sellaista koodia ei pääse syntymään, jossa samankaltaisuusoperaattorit mahdollisesti suorittaisivat tyyppimuunnoksia ennen vertailua ja samalla vältytään vaarallisilta olettamuksilta lopputuloksesta.

Automaattinen puolipisteen sijoitus

Automaattisen puolipisteen sijoittamisen mahdollisilta ongelmilta vältytään myös TypeScriptin staattisen koodianalyysin ansiosta. Jos tarkastellaan

koodiesimerkissä 3.1 esiteltyä ongelmaa ja yritetään toistaa sama virheellinen toiminnallisuus TypeScriptissä, ilmoittaa staattinen koodianalyysi virheestä. Koodiesimerkissä 4.6 määritellään funktion toivottu palautusarvo rajapinnan avulla ja yritys asettaa `return`-lauseen jälkeinen aaltosulku seuraavalla riville johtaa virhetilanteeseen, joka on korjattava, ennen kuin koodi voidaan kääntää JavaScript-koodiksi.

```
1  interface Foo {
2      foo: string;
3  }
4
5  function fooFunc(): Foo {
6      return
7      {
8          foo: 'bar',
9      }
10     // Error: Expression expected.
11 }
```

Koodiesimerkki 4.6: Automaattinen puolipisteen sijoitus ei pääse aiheuttamaan ongelmaa, koska TypeScript ei salli koodin kääntämistä virheellisessä muodossa

Prototyyppipohjainen periytyminen

TypeScript tarjoaa ratkaisun prototyyppipohjaiseen periytymiseen liittyviin ongelmiin lisäämällä tuen luokkia, nimiavaruuksia, moduuleja ja rajapintoja varten. Tämä mahdollistaa, että ohjelmoijat voivat hyödyntää jo mahdollisesti muiden kielien kautta oppimiaan taitoja liittyen olioihin ja koodin rakenteeseen, mukaan lukien rajapintojen hyödyntämisen ja luokkien pe-

rimisen. Luokat ja moduulit ovat osana ECMAScript 6 -määrittelyä, ja koska TypeScript-koodi on mahdollista kääntää yhteensopivaksi myös aikaisempien ECMAScript-versioiden kanssa, voi ominaisuuksia hyödyntää jo tässä vaiheessa, vaikka kohdealusta ei vielä tukisikaan kyseisiä ominaisuuksia sellaisenaan.

Automaattinen tyyppimuunnos

TypeScript ratkaisee luvussa 3.6 havainnollistetut automaattiseen tyyppimuunnokseen liittyvät ongelmat luvussa 4.2 esitellyn tyyppitarkistuksen avulla. Tyyppitarkistuksen avulla kääntäjä voi jo kehitysvaiheessa antaa varoituksia tahattomaan automaattiseen tyyppimuunnokseen liittyen ja auttaa korjaamaan ongelmat, ennen kuin ne edes pääsevät syntymään. Tyypityksen käyttö TypeScriptissa ei estä dynaamisen tyypityksen käyttöä, vaan kehittäjä voi itse vaikuttaa, milloin haluaa käyttää ominaisuutta ja milloin käyttää dynaamista käyttäytymistä mahdollistavaa `any`-tyyppiä [Fen18, s. xxiv].

Näkyvyysalueet

TypeScriptin avulla ECMAScript 6 -versioon lisätyt lohkonäkyvyyteen sidotut uudet muuttujatyypit `const` sekä `let` ovat käytettävissä riippumatta kohde-JavaScript-versiosta ja niiden avulla vältytään jo suurelta osalta luvussa 3.6 esitetyiltä näkyvyysalueisiin liittyviltä ongelmilta. Lisäksi TypeScriptin staattinen koodianalyysi ilmoittaa virheestä, mikäli muuttujan alustuksesta puuttuu avainsana, joten sen myötä on mahdotonta ylentää muuttujaa vahingossa globaaliin näkyvyysalueeseen. Sulkeumien käyttö on edelleen mahdollista, vaikka käytettäisiinkin TypeScriptiä, eli JavaScriptin hyviä ominaisuuksia ei suljeta pois [Fen18, s. xxv].

Valinnainen staattinen tyypitys

Luvun 3.6 viimeiseksi esitettynä JavaScriptiin liittyvänä puutteena mainittiin dynaaminen tyypitys. TypeScript sisältää valinnaisen staattisen tyypityksen, joka puolestaan tuo mukanaan useita hyötyjä. Tyypitys on valinnaisesti staattinen, koska tarvittaessa on mahdollista käyttää `any`-tyyppiä, joka sallii tyypin dynaamisen käyttäytymisen.

Staattisen tyypityksen on osoitettu parantavan sekä koodin laatua että ymmärrettävyyttä. Erityisesti koodin refaktoroinnin yhteydessä staattisesta tyypityksestä on hyötyä. Sen ansiosta mahdolliset virheet voidaan huomata jo käännösvaiheessa sen sijaan, että ne ilmenisivät vasta suoritusaikaisesti. Muodolliset tyyppitiedot toimivat myös erinomaisena koodin dokumentaationa, joka pysyy aina ajan tasalla. Tyyppitiedoista voidaan nopeasti päätellä esimerkiksi, minkä tyyppisiä arvoja funktiokutsu hyväksyy ja minkä tyyppisen arvon funktio palauttaa [Bas19].

Muodollisen tyyppitiedon ja käännösaikaisen tyyppitarkistuksen ansioista voidaan lisäksi yleensä luottaa siihen, että operaatioita tullaan kutsumaan oikealla tavalla myös suoritusaikaisesti tarvitsematta rakentaa jokaista operaatiota varten kalliita testejä [Car04, s. 5].

4.4 Vaihtoehtoiset ratkaisut

TypeScript ei ole ainoa vaihtoehto tavallisen JavaScriptin kirjoittamiselle, vaan muitakin vaihtoehtoja on olemassa, kuten esimerkiksi *Babel*, *CoffeeScript*, *Dart* ja *Flow*.

Vahvana vaihtoehtona TypeScriptille on **Babel**; kääntäjä, joka mahdollistaa uusimpien ECMAScript-toimintojen käyttämisen nykyisissä selaimissa suorittamalla koodin kääntämisen vanhempaan ECMAScript-versioon toiminnallisuuksia karsimatta. Hankkeen tavoitteena on saada uusimmat

ECMAScript-ominaisuudet käyttöön paljon nopeammin kuin mitä muuten olisi mahdollista, koska usein esiintyy viiveitä, ennen kuin selaimet alkavat tukea uusimpia ominaisuuksia [BAB19]. Suurimpana erona TypeScriptiin on, että Babel ei tarjoa käännösaikaista tyyppitarkistusta.

CoffeeScript oli useiden vuosien ajan suosittu vaihtoehto tavalliselle JavaScriptille. Kielessä on lyhytsanainen syntaksi ja kieli käännetään lopulta suoritusta varten tavalliseksi JavaScript-koodiksi. CoffeeScript ei tarjo monia niistä lisäominaisuuksista, joita TypeScript tarjoaa, kuten esimerkiksi staattinen tyyppitarkistus. Kieli eroaa lisäksi hyvin paljon JavaScriptistä, joten sen käyttö vaati uuden ohjelmointikielen opettelun. Vuonna 2018 CoffeeScript saavutti kolmannen sijan Stack Overflown suorittamassa "kauheimmat kielet"-kehittäjäkyselyssä (engl. *most dreaded languages*) [Sta18] ja sen suosio on koko ajan hiipumassa.

Dart on Googlen kehittämä ohjelmointikieli, ja sillä on paljon yhteistä TypeScriptin kanssa. Se on luokkapohjainen, olioperustainen ja tarjoaa myös valinnaisia tyypityksiä, jotka voidaan tarkistaa staattisen tyyppitarkistuksen avulla. Dart-kieltä suunniteltiin alun perin JavaScriptin korvaajaksi, ja se oli tarkoitus kääntää JavaScript-koodiksi vain sen takia, jotta kieltä tuettaisiin laajalti myös sillä välin, kunnes kielen natiivituki saataisiin lisättyä selaimiin. Nykyään on mahdollista valita, jos kieli halutaan kääntää JavaScriptiksi tai jos kieltä halutaan ajaa sellaisenaan Dart software development kitin avulla. [Dar19]

Myös Facebook on kehittänyt JavaScriptille oman staattisen tyyppitarkistustyökalunsa, joka kulkee nimellä **Flow** [Fac20]. Flow tarjoaa TypeScriptin tavoin tyyppipäättelyä ja osaa antaa kehityksenaikaista palautetta mahdolliseen tyyppien vääriinkäyttöön liittyen.

5 Aikaisempi tutkimus ja näkökulmia

Tässä luvussa tutustutaan tutkielman aiheeseen liittyvään aikaisemmin suoritettuun tutkimukseen ja näkökulmiin.

5.1 Staattinen vai dynaaminen tyypitys

Meijer ja Drayton tutkivat vuonna 2005, olisiko staattinen vai dynaaminen tyypitys parempi lähtökohta ohjelmointikielelle [MD04]. Tutkimus suoritettiin tutkimalla aiheeseen liittyvää kirjallisuutta sekä haastattelemalla alan asiantuntijoita.

Lopputulokseksi saatiin, että staattinen tyypitys on tehokas työkalu, joka auttaa kehittäjiä ilmaisemaan olettamuksia ongelmasta, jota he yrittävät ratkaista, ja antaa heille mahdollisuuden kirjoittaa ytimekkäämpää ja virheettömämpää koodia. Lisäksi todettiin, että epävarmojen olettamusten, dynaamisuuden ja muutosten käsittely on yhä tärkeämpää hajautetussa maailmassa. Tutkimuksessa todettiin, että sen sijaan, että keskityttäisiin dynaamisesti ja staattisesti tyypitettyjen kielten eroihin, tulisi kesittyä saamaan dynaaminen ja staattinen tyypitys rauhanomaisesti yhdistettyä samaan kieleen. Loppulos tiivistettiin seuraavaan lauseeseen [MD04]:

"*Static typing where possible, dynamic typing when needed!*"

Joka vapaasti käännettynä tarkoittaa, että staattista tyypitystä tulisi käyttää aina, kun se on mahdollista, ja dynaamista tyypitystä silloin kun, se on tarpeellista.

5.2 JavaScript-ohjelmointivirheiden syyt ja seuraamukset

Ocariza Jr., Bajaj, Pattabiraman & Mesbah suorittivat vuonna 2017 tutkimuksen, jossa tutkittiin JavaScript-ohjelmointivirheiden syitä ja seuraamuksia [OBPM17]. Tutkimuksen tavoitteena oli ymmärtää JavaScript-ohjel-

mointivirheiden perimmäisiä syitä sekä vaikutuksia ja miten tulokset voivat vaikuttaa JavaScript-ohjelmoijiin, testaajiin ja työkalujen kehittäjiin. Tutkimus oli empiirinen, ja siinä tutkittiin 502 ohjelmointivirhettä 19 eri tietolähteestä. Virheisiin liittyviä virheilmoituksia tutkittiin perusteellisesti, jotta saatiin selville jokaisen virheen syy sekä seuraamus ja jotta ne pystyttiin luokittelemaan.

Tuloksista selvisi, että 33 % virheistä liittyivät tavalla tai toisella tyypityksiin. Tuloksista selvisi myös, että peräti 75 % kaikista virheistä johtuivat "*Incorrect Method Parameter*" -tyyppisestä virheestä, joka tarkoittaa, että JavaScript-metodia oli kutsuttu odottamattomalla tai virheellisellä arvolla. Esimerkkinä "*Incorrect Method Parameter*" -tyyppisestä virheestä mainitaan tapaus, jossa JavaScriptin `Date`-olion `setDate()`-metodia on kutsuttu merkkijonoarvolla, kun metodi odottaa sitä kutsuttavan kokonaisluvulla.

Virheiden seuraamuksista todettakoon vielä, että tutkimus osoitti 57 % virheistä johtavan koodin suorituksen päättävään poikkeukseen.

5.3 Staattisten tyyppijärjestelmien avulla havaitut virheet

Vuonna 2017 Zheng Gao, Christian Bird & Earl T. Barr suorittivat yhdessä tutkimuksen [GBB17], jossa he yrittivät selvittää, minkälaisia hyötyjä staattisten tyyppijärjestelmien avulla voisi saavuttaa JavaScript-kehityksessä. Tyyppijärjestelmien puolesta heidän tutkimukseensa valittiin kaksi merkittävää alan järjestelmää: Microsoftin kehittämä *TypeScript 2.0* [Mic19] sekä Facebookin kehittämä *Flow 0.30* [Fac20]. Heidän tutkimuskysymyksensä keskittyivät siihen, millaisia vaikutuksia tyyppijärjestelmillä on koodin laatuun ja tarkemmin moneltako ohjelmointivirheeltä oltaisiin voitu välttyä käyttämällä tyyppijärjestelmiä.

Tutkimus suoritettiin tarkastelemalla korjattuja virheitä julkisesti saa-

tavilla olevista tietolähteistä, lisäämällä tyyppimerkintöjä korjausta edeltävään, edelleen ohjelmointivirheen sisältävään versioon, ja tarkistamalla jos TypeScript ja Flow osaisivat ilmoittaa tiedossa olevasta virheestä. Jos tyyppijärjestelmät osasivat ilmoittaa virheestä, oletettiin, että kehittäjä olisi korjannut virheen eikä se olisi koskaan päätynyt koodivarastoon (engl. *code repository*) saakka. Lopuksi vertailtiin tyyppijärjestelmien ilmoittamien virheiden osuutta kaikkiin tiedossa oleviin virheisiin.

Tutkimuksen lähdemateriaalina käytettiin 398 eri projektia ja lopulta tutkimukseen otettiin mukaan 400 ohjelmointivirhettä kyseisistä projekteista. Tutkimuksen lopputulokseksi saatiin, että sekä TypeScriptin että Flown avulla oltaisiin havaittu 15 % tutkituista virheistä jo ohjelmien kehitysvaiheessa.

6 Tutkielmassa suoritettava tutkimus

Tässä luvussa tarkastellaan tutkielman tutkimusongelmaa ja tavoitetta sekä perustellaan tutkimuksen tarpeellisuutta. Luvun lopuksi esitellään tutkimusmenetelmää, tutkimuksen suunnitteluun vaikuttavia tekijöitä, tutkimuksen validiteettiuhkia sekä asioita, jotka vaikuttavat kyselytutkimuksen kysymysten muotoiluun.

Tutkimuksen tarkoituksena on selvittää web-ohjelmistoalan yrityksissä työskentelevien asiantuntijoiden suhtautumista TypeScriptiin ja sen hyötyihin ja haittoihin liittyen JavaScript-ohjelmistokehitykseen sekä vastaavatko asiantuntijoiden mielipiteet aikaisempia tieteellisiä tutkimustuloksia.

Tutkielman tutkimusmenetelmänä on kyselytutkimus, joka toteutetaan Internetissä suoritettavan kyselylomakkeen avulla.

6.1 Tutkimusongelma

Tutkimusongelmana on, miten hyödyllisenä tai haitallisena TypeScriptiä pidetään web-ohjelmistoteollisuuden yrityksissä ja millä ohjelmistokehitysprosessin osa-alueilla hyödyt ja haitat korostuvat.

Tutkimusongelmaan liittyvää tieteellistä tutkimusta on tarjolla jonkin verran, ja se on pääosin sitä mieltä, että TypeScript tuo mukanaan paljon hyötyjä ja sitä kannattaa käyttää perinteisen JavaScriptin sijasta. Aiheeseen liittyvistä tutkimuksista ei selkeästi ilmene, kokevatko kehittäjät itse TypeScriptin hyödyllisenä ja missä kohtaa ohjelmistokehitysprosessia siitä koetaan olevan eniten hyötyä. TypeScriptin mahdollisiin haittapuoliin liittyen ei juurikaan löydy minkäänlaista tieteellistä tutkimusta. Tutkimuksista ei myöskään ilmene, jos TypeScriptille olisi olemassa jokin vielä parempi vaihtoehto.

Tavoitteena on kerätä tietoa web-ohjelmistokehitystä tekevien yritysten

ohjelmointikielimieltymyksistä sekä asiantuntijoiden näkemyksiä TypeScriptiin liittyvistä hyödyistä ja mahdollisista haitoista.

Tavoitteen saavuttamiseksi pyritään kyselytutkimuksen avulla löytämään vastauksia seuraaviin tutkimuskysymyksiin:

- **TK1:** *Miten hyödylliseksi ohjelmointikieleksi TypeScript koetaan?*
- **TK2:** *Millä ohjelmistokehityksen osa-alueilla TypeScript tarjoaa eniten hyötyjä?*
- **TK3:** *Koetaanko TypeScriptin käyttöön liittyvän haittoja?*
- **TK4:** *Mille ohjelmistokehityksen osa-alueille TypeScriptin haitat keskittyvät?*
- **TK5:** *Millä muulla tavalla JavaScriptiin liittyvät ongelmat ja puutteet on ratkaistu, jos TypeScript ei ole käytössä?*

6.2 Tutkimuksen tarpeellisuus

Web-ohjelmistokehittäjien keskuudessa on olemassa yleinen vallitseva käsitys, että TypeScript tuo mukanaan paljon hyötyjä JavaScript-ohjelmistokehitykseen, sen käyttö helpottaa sovelluksen ylläpitoa ja sen avulla voidaan jopa vähentää yksikkötestien määrää. Onko asia todellisuudessa näin, vai onko kyseessä vain hetkellinen ylimainostus?

Vallitsevan käsityksen tueksi on olemassa rajoitetusti tieteellistä tutkimusta, eikä siinä varsinaisesti oteta kantaa, mihin web-ohjelmistokehityksen osa-alueisiin TypeScriptin hyödyllisyys liittyy. TypeScriptin mahdollisia haittapuolia ei lisäksi ole juuri lainkaan tutkittu, eikä TypeScriptin hyötyjä tai haittoja käsittelevää tieteellistä tutkimusta ole lainkaan saatavilla suomeksi.

Tämän vuoksi on hyödyllistä saada kartoitus web-ohjelmistokehitystä ammattimaisesti harjoittavien yritysten asiantuntijoiden mielipiteisiin ja suhtautumiseen TypeScriptiin.

6.3 Tutkimusmenetelmä

Tutkielman tutkimusmenetelmäksi valittiin kyselytutkimus, koska kyselylomakkeen avulla on mahdollista saada paljon kvalitatiivista tietoa nopeasti ja sen avulla vastaukset saadaan suoraan strukturoidussa muodossa, josta niitä on tehokasta käsitellä. Kyselylomake on kirjallinen kysymyslista, johon vastaaminen tapahtuu Internetissä.

Internetissä suoritettava kyselylomake on kyselyyn vastaajille mielekäs, koska he voivat vastata kyselyyn heille sopivana ajankohtana, osallistuminen on mahdollista toteuttaa anonyymisti [VJ06, s. 437] eikä vastaajien maantieteellinen sijainti toimi vastaamisen rajoitteena [SSS08, s. 15]. Kyselylomakkeen avulla poistetaan myös mahdollinen haastattelijan puolueellisuus, joten saatujen vastausten tulisi kuvastaa vastaajien todellisia mielipiteitä [VJ06, s. 438].

Tutkimuksen suunnittelu

Erään tutkimuksen [SR07, s. 9-10] mukaan Internetissä suoritettaviin kyselylomakkeisiin vastaa keskimäärin 30 % henkilöistä, jotka ovat saaneet kyselyn ja keskimääräinen vastausaika on alle kaksi viikkoa. Toisen tutkimuksen [SSS08, s. 15] mukaan ohjelmistokehitystä koskevien kyselylomakkeiden tapauksessa vastaava vastausprosentti on ainoastaan 5 %, joten on tärkeää tavoittaa suuri määrä oikeanlaisia potentiaalisia vastaajia, jotta lopullinen kyselyyn vastanneiden määrä on tarpeeksi suuri, jotta tuloksien perusteella voidaan tehdä oikeita johtopäätöksiä. Riittävä määrä vastauksia olisi 30–500

kappaletta [SR07, s. 32-33].

Tutkielmassa käytettävän kyselylomakkeen vastaajiksi soveltuvat parhaiten henkilöt, jotka ovat joko aikaisemmin työskennelleet tai työskentelevät parhaillaan ohjelmistokehityksen parissa ja joilla on omakantaista kokemusta sekä JavaScript- että TypeScript-ohjelmoinnista. Kyselylomake toteutetaan englanniksi, joten sujuva englanninkielentaito on myös edellytys vastaajille.

Vastaajien löytämiseksi sovelletaan **lumipallo-otantaa** (engl. *snowball sampling*) ja hyödynnetään olemassa olevia suhdeverkostoja. Lumipallo-otanta perustuu siihen, että ensin tunnistetaan yksi osallistuja, joka täyttää osallistumiskriteerit. Tätä ensimmäistä osallistujaa pyydetään seuraavaksi suosittelemaan toista osallistujaa; sitten toista osallistujaa pyydetään suosittelemaan seuraavaa osallistujaa ja niin edelleen. Menetelmä toimii parhaiten pienissä väestöryhmissä, joissa jäsenet tuntevat toisensa, ja se soveltuu erinomaisesti määritellyn ja erittäin kohdennetun kohderyhmän tapauksessa [SR07, s. 32].

Tavoitteena on tunnistaa yrityksiä, jotka työskentelevät web-ohjelmistokehityksen parissa, ja tunnistaa kyseisistä yrityksistä avainhenkilöitä, joiden kautta kyselyt saadaan jaettua yrityksessä työskenteleville osallistumiskriteerit täyttäville henkilöille. Esimerkiksi `http://www.itewiki.fi`-sivuston kautta on mahdollista hakea suomalaisia it-alan yrityksiä osaamisalueittain.

Kyselylomake toteutetaan Helsingin yliopiston e-lomakkeen avulla, koska *helsinki.fi*-verkkotunnus kielii luotettavasta verkko-osoitteesta ja palvelun kautta vastaukset eivät päädy oppilaitoksen ulkopuolisille tahoille. E-lomake on suoraan kaikkien yliopistolaisten käytettävissä oleva Opetusteknologiapalvelujen tukema verkkolomakepalvelu, jonka avulla voidaan luoda räätälöityjä kyselyjä, joille voidaan asettaa voimassaoloaika ja saada suora julkinen linkki. Kyselylomakkeen voimassaoloajaksi asetetaan kaksi viikkoa. Kyselyn linkin

lomassa toimitettu saatekirje sekä kyselyn vastaamisajan loppupuolella lähetetty muistutusviesti voivat nostaa kyselyn vastausprosenttia [VJ06, s. 443, 447-448], joten molemmat pyritään toteuttamaan tämän kyselyn yhteydessä.

Kyselylomakkeen kysymykset

Kyselylomakkeen avulla esitetty kysymys on mittaustyökalu, jonka avulla tutkija voi saada selville vastaajan mielipiteen, tiedot ja käyttäytymisen. Asianmukaisesti laaditut kysymykset ovat välttämättömiä kaikille kyselyille, ja kaikilla hyvillä kyselykysymyksillä on tiettyjä yhteisiä ominaisuuksia. Parhaat kyselykysymykset ovat lyhyitä, yksiselitteisiä ja merkityksellisiä vastaajalle [SSS08, s. 66-68].

Tutkielman tutkimusongelman kvalitatiivisen luonteen vuoksi kyselylomakkeella on pääosin avoimia kysymyksiä. Avoimet kysymykset ovat erityisen hyödyllisiä tutkittaessa uusia aiheita, ja ne tarjoavat mahdollisuuden oppia odottamattomia tietoja. Lisäksi avointen kysymysten avulla saadaan yleensä pätevämpiä tuloksia kuin suljettujen kysymysten avulla, koska silloin vastaajien ei ole pakko valita tutkijan luomista vastausvaihtoehdoista [SSS08, s. 71].

Jokaisen kyselylomakkeella esitetyn kysymyksen tulisi liittyä tavalla tai toisella tutkimuksen tavoitteeseen tai tutkimusongelmaan. Kysymykset pyritään muotoilemaan mahdollisimman neutraaliin muotoon, jotta ne eivät johdattelisi vastaajia vastaamaan tietyllä tavalla, ja muotoilussa on myös otettava huomioon, että kyselyyn vastaavat henkilöt ymmärtävät kysymyksen sen laatijan tarkoittamalla tavalla. Tarvittaessa lisätään ohjeet kysymyksiin vastaamista varten, ja kyselyn kokonaispituus pyritään pitämään mahdollisimman lyhyenä, jotta kyselyyn vastaamista ei keskeytettäisi sen pituuden takia kesken [VJ06, s. 441].

Tutkimusongelman kvalitatiivisuuden vuoksi tämän tutkielman asiayhteydessä ei ole tarpeen vertailla avointen kysymysten tuloksia tilastollisesti keskenään, vaan tarkoitus on analysoida, miten ne vertautuvat olemassa olevaan tutkimukseen. Vastaajilta tullaan kyselyn lopuksi kysymään myös yleisluontoisia demografiatietoja, jotta vastauksia voidaan tarpeen tulleen ryhmitellä yhteen.

Lisäksi, jotta saadaan yleiskuva siitä, kuinka hyödyllisenä TypeScriptiä pidetään, vaikka avointen kysymysten vastaukset sattuisivatkin jäämään tyngiksi, hyödynnetään yritysmaailmassa usein asiakasuskollisuuden mittaamiseen käytettyä NPS-mittaria (eng. *Net Promotor Score*), joka tiivistettynä vastaa kysymykseen: *"Kuinka todennäköisesti suosittelisit palvelua ystävällesi tai kollegallesi?"*. NPS-kysely on helppo tulkita ja toteuttaa, ja lisäksi suositteluihin pohjautuva kysymys on osoittautunut olevan tehokas käytöksen ennustaja. NPS-kyselyssä vastaajia pyydetään antamaan vastauksensa asteikolla 0-10 ja vastaukset jaetaan sen jälkeen kolmeen kategoriaan: arvostelijat, passiiviset ja suosittelijat. Arvostelijoihin lasketaan vastaukset väliltä 0-6, passiivisiin vastaukset väliltä 7-8 ja suosittelijoihin vastaukset väliltä 9-10. Kun vastaukset on luokiteltu, lasketaan lopulliset NPS-pisteet seuraavalla kaavalla:

$$NPS = \frac{(\text{suosittelijoiden määrä} - \text{arvostelijoiden määrä})}{(\text{vastaajien määrä})} \times 100$$

Mikäli NPS-pisteiden tulos on yli 0, voidaan ajatella, että suurin osa vastaajista suosittelee palvelua. Yli 50 on jo puolestaan erinomainen tulos, ja yli 70 jo maailmanluokkaa [Sal19].

Validiteettiuhat

Kyselykysymykset ovat päteviä siinä määrin, kuin ne mittaavat tutkittavia taustakäsitteitä. Kyselylomake ei ole itsessään kelvollinen tai virheellinen, vaan kyselyn pätevyyttä voidaan arvioida vain tutkimalla kysymysten ja asenteiden välistä suhdetta, käyttäytymistä tai tosiasiaa, jota kyselyllä aiotaan mitata. Yksinkertaisesti sanottuna kelvolliset kysymykset mittaavat, mitä niiden pitäisi mitata [SR07, s. 40].

On useita syitä, joiden vuoksi tutkimukseen osallistujat voivat antaa virheellisiä tietoja vastatessaan kyselyyn. He voivat esimerkiksi tietoisesti antaa vääriä tietoja välttääkseen nolostumista tai jotta annettu vastaus olisi enemmän sosiaalisesti hyväksyttyjen normien mukainen. Tätä validiteettiuhkaa voidaan huomattavasti pienentää korostamalla, että vastaukset annetaan nimettömästi ja että vastauksia ei voida yhdistää tiettyyn vastaajaan. Kysymysten asettelussa on myös mahdollista esittää sosiaalisesta normistosta poikkeavia tutkimustuloksia, jotta vastaaja tuntee turvallisemmin voivansa antaa oman aidon mielipiteensä.

Toinen teoreettinen uhka kyselyn validiteetille on, jos kysytään sellaisia asioita tai yksityiskohtia, joita voi olla vaikea muistaa tai joita vastaaja ei pysty tarkasti arvioimaan. Uhkana on myös, että vastaajat voivat tarjota mielipiteitä kyselyihin vain siksi, että joku pyytää heiltä mielipidettä, eikä siksi, että heillä todella on sellainen [SR07, s. 40].

Kyselyn pätevyys ja siihen liittyvä mittauksen luotettavuuden tai johdonmukaisuuden käsite voi myös olla uhattuna kyselyn kysymysten sanamuodon ollessa virheellinen tai jos kysymysten vastausvaihtoehdot ovat puutteellisia tai epäasianmukaisia. Tällaisilta uhilta voidaan välttyä suorittamalla kyselyn esitestaus ja tehden sen tulosteiden perusteella asianmukaiset korjaukset. Esitestauksen avulla voidaan muun muassa varmistua siitä, että kysymykset

ovat ymmärrettäviä, sekä arvioida kyselyn luotettavuutta ja pätevyyttä. Kyselyn esitestauksen haittapuoli on, että esitestaukseen osallistuneet vastaajat eivät voi osallistua lopulliseen kyselyyn [SSS08, s. 77-78].

Mahdollisesti suurin uhka kyselylomakkeen validiteetille on sen avulla saatava verrattain pieni otanta, jonka perusteella ei mahdollisesti voida tehdä yleispäteviä johtopäätöksiä. Vastaajien määrää on mahdollista kasvattaa liittämällä kyselyn linkin mukaan saateviesti, jossa selitetään, mitä kyselyllä haetaan takaa, ja lähettämällä alkuperäisen viestin lisäksi muistutusviesti ennen vastausajan päättymistä. Otannan riittävyyden ja vastausten yhdenmukaisuuden varmistamiseksi voidaan hyödyntää puolijakoanalyysiä (engl. *split-half analysis*). Menettelyyn kuuluu, että vastaukset jaetaan satunnaisesti kahteen osaan, jonka jälkeen osat analysoidaan erikseen. Jos eri osien tulokset ovat yhdenmukaisia, on vastauksia kerätty tarpeeksi; jos tulokset eroavat suuresti toisistaan, on vastauksia kerättävä lisää [SR07, s. 33].

7 Kyselytutkimuksen suorittaminen

Tässä luvussa kerrotaan tutkimuksen suorittamisen eri vaiheista. Ensimmäisessä aliluvussa käsitellään kyselyn valmistelu, toisessa aliluvussa käydään läpi kyselyn suorittamiseen liittyvät asiat ja viimeiseksi kerrotaan tulosten analysointimenetelmästä.

Kyselytutkimus suoritettiin 19.11.2019 – 4.12.2019 välisenä aikana. Vastauksia saatiin ainakin kymmenestä eri yrityksestä. Kuudessa vastauksessa yritystieto jätettiin antamatta. Yksittäisiä vastauksia saatiin yhteensä 21 kappaletta, minkä vuoksi oli tärkeää käydä vastaukset läpi kertaalleen ja yrittää ryhmitellä samankaltaiset vastaukset yhteen.

7.1 Kyselytutkimuksen valmistelu

Kyselylomakkeet koostuvat kirjallisesti esitetyistä kysymyksistä, ja ne ovat yleinen tutkimustekniikka, koska niitä voidaan suorittaa nopeasti ja helposti. Monet tekijät, kuten kysymysten sanamuoto, lomakkeen asettelu ja kysymysten järjestys, voivat kumminkin vaikuttaa tutkimuksen tuloksiin, joten niihin on kiinnitettävä erityistä huomiota [SSS08].

Jotta voitiin varmistua siitä, että kyselyn kysymykset olisivat kaikin puolin ymmärrettäviä ja että kysymysten järjestys on looginen, suoritettiin kyselylle esitestaus. Kyselyn kysymyksiä sekä saatekirjeen sisältöä esitestattiin kerran kahden henkilön toimesta ennen varsinaisen kyselylomakkeen julkaisemista. Esitestauksen perusteella kysymysten muotoiluun tehtiin pientä hienosäätöä ja yksi kysymys jaettiin kahteen eri kysymykseen. Saatekirjeeseen päätynyt kirjoitusvirhe huomattiin myös esitestauksen yhteydessä, joten sekin korjattiin tässä yhteydessä. Myös johdantotekstiä selkeytettiin hieman esitestauksen jälkeen. Helsingin yliopiston e-lomakkeen avulla luotua kyselylomaketta testattiin myös kerran yhden henkilön toimesta, jotta voitiin

varmistua että lomakkeen tallennus toimi teknisessä mielessä ja että vastaukset tallentuivat tarkoitetulla tavalla. Lomakkeen esitestauksen päätteeksi testivastaukset poistettiin tietokannasta.

7.2 Kyselytutkimuksen suorittaminen

Kyselytutkimus suoritettiin Helsingin yliopiston e-lomakkeen avulla, koska e-lomake on *helsinki.fi*-alidomainissa ja sen myötä pyrittiin luomaan vastaajille mielikuva luotettavasta kyselystä sekä uskottavasta tutkimuksesta.

E-lomakkeen saatavilla olevan ohjeen ansiosta kyselyn luominen osoittautui melko vaivattomaksi tehtäväksi. Kysymysten vastaukset eivät vaikuttaneet toisiin kysymyksiin ja kaikki kysymykset luotiin samalle sivulle, joten kysymyksille ei ollut tarvetta luoda mitään näkyvyyssääntöjä, joka helpotti kyselyn rakentamista. Kyselyn luontivaiheessa ainoaksi varsinaiseksi ongelmaksi osoittautui se, että kysymysten luontinäkymässä kenttä, johon toivottu kysymysteksti tuli syöttää, oli rajatun kokoinen, joten syötettyä tekstiä ei pystynyt kerralla näkemään kokonaisuudessaan. Kirjoitusvirheiden välttämiseksi kelpo ratkaisuksi osoittautui kirjoittaa kysymysteksti tekstieditoriin, josta sen pystyi tietokoneen leikepöydän avulla siirtämään muuttumattomana sille tarkoitettuun kenttään kysymyksen luontinäkymässä.

Kyselylomakkeen saatekirje, joka sisälsi linkin kyselylomakkeelle, lähetettiin sähköpostitse 62 web-ohjelmistokehitystä ammattimaisesti harjoittavan yrityksen yhteyshenkilölle, ja saatekirjeessä pyydettiin jakamaan kyselylomakkeen linkkiä eteenpäin kaikille mahdollisiksi vastaajiksi soveltuville henkilöille yrityksessä. Saatekirjeessä selitettiin, että vastaajiksi soveltuvat parhaiten henkilöt, jotka ovat joko aikaisemmin työskennelleet tai työskentelevät parhaillaan ohjelmistokehityksen parissa ja joilla on omakantaista kokemusta sekä JavaScript- että TypeScript-ohjelmoinnista. Yhteyshenkilöiden yhteys-

tiedot saatiin *http://www.itewiki.fi/*-verkkosivun kautta. Itewiki-sivustolta pystyi etsimään kyselyn kannalta olennaisia yrityksiä tehokkaasti ja yritysten tiedoissa oli myös yhteyshenkilöiden yhteystiedot. Muutaman yrityksen osalta yhteystietoja ei ollut saatavilla, joten ne yritykset karsittiin pois kyselyn vastaanottajista.

Kyselyn vastausajaksi asetettiin 16 päivää. Kyselyä suoritettaessa huomattiin, että vastauksia tuli ainoastaan muutaman ensimmäisen vuorokauden kuluessa yhteydenotosta, mutta tämän jälkeen vastausten tuleminen loppui kokonaan. Kun vastausaikaa oli jäljellä viikko, oli vastauksia saatu 16 kappaletta. Siinä vaiheessa Itewiki-sivuston kautta haettiin lisää yrityksiä, joiden työntekijät voisivat soveltua vastaajiksi. Yrityksiä löytyi 26 lisää, joten kyselylomakkeen saatekirje lähetettiin myös näiden yritysten yhteyshenkilöille sähköpostitse. Uusien yhteydenottojen ansiosta vastauksia saatiin vielä viisi lisää.

Tutkielman tavoitteena oli saada vähintään 30 vastausta, mutta valitettavasti lopullinen 21 vastauksen määrä jäi tavoitteesta hieman vajaaksi. Yhteensä saateviestejä lähetettiin 88 kpl, joten vastausprosentiksi jäi 23,9 %.

7.3 Vastausten käsittely

Vastauksia tuli yhteensä 21 kappaletta ainakin ainakin kymmenen eri yrityksen edustajilta. Kuusi vastaajista jätti vastaamatta kysymykseen työnantajasta. Yksi vastaajista kertoi vastauksissaan, ettei hänellä ollut lainkaan kokemusta TypeScriptistä, joten kyseisen vastaajan vastaukset karsittiin pois kaikista jatkoanalyyseistä, koska vastaaja ei täyttänyt vastaajien soveltuvuusvaatimuksia. Pääosa kyselyn kysymyksistä oli avoimia, ja niiden tarkoituksena oli löytää ilmiöitä ja kerätä tietoa TypeScriptin hyödyllisyydestä JavaScript-ohjelmistokehityksessä. Koska kysymykset olivat avoimia

ja tutkimusongelmaa tarkasteltiin vastaajien omien mielipiteiden ja kokemusten pohjalta, käsiteltiin vastaukset temaattisen synteesin (engl. *thematic synthesis*) avulla [CD11].

Temaattinen analyysi (engl. *thematic analysis*) on lähestymistapa, jota usein käytetään kvalitatiivisessa tutkimuksessa yhteisten kaavojen tunnistamiseen, analysointiin ja raportointiin. Temaattinen synteesi perustuu temaattisen analyysin perusperiaatteisiin, ja sen avulla voidaan tunnistaa toistuvia teemoja tai aiheita useista tutkimuksista, tulkita ja selittää kyseisiä teemoja sekä tehdä johtopäätöksiä systemaattisella tavalla [CD11].

Temaattisen synteesin periaatteiden mukaisesti ensimmäinen askel oli käydä läpi kyselyn kysymykset ja vastaukset kerran, jotta pystyi luomaan käsityksen koko aineiston laajuudesta ja syvyydestä. Koko ainaiston läpikäynnin perusteella oli jo mahdollista tunnistaa alustavia kaavoja ja ideoita tuloksista, joten se helpotti tulevia vaiheita. Läpikäynnin jälkeen oli vuorossa mielenkiintoisten havaintojen systemaattinen tunnistaminen sekä nimiöinti. Sen jälkeen samankaltaiset nimikkeet yhdistettiin yhteisiksi teemoiksi ja teemoille annettiin otsikot. Kun teemat oli tunnistettu, oli vuorossa teemojen välisten yhteyksien analysointi, jotta pystyttiin tunnistamaan, mitkä teemat liittyivät toisiinsa. Lopullisten teemojen ja niiden välisten yhteyksien avulla tuloksista pystyi muodostamaan selkeän kokonaiskuvan ja saatiin aikaiseksi temaattinen synteesi.

Viimeisenä vaiheena suoritettiin synteesiin johtavien tulkintojen luotettavuuden arviointi. Arviointi suoritettiin vertailemalla synteesiä alkuperäisiin vastauksiin, jotta pystyttiin varmistumaan siitä, ettei missään vaiheessa olla vedetty vääriä johtopäätöksiä ja että kaikille tunnistetuille teemoille löytyy perustelu.

8 Analyysi kyselytutkimuksen tuloksista

Tässä aliluvussa esitellään tämän tutkielman kyselytutkimuksen tulokset. Ensimmäisessä aliluvussa kuvataan kyselytutkimuksen rakenne ja perustellaan, millä tavalla kysymykset liittyvät tutkimusongelmaan. Toisessa ja kolmannessa aliluvuissa analysoidaan TypeScriptin hyödyllisyyteen liittyviä vastauksia. Neljännessä aliluvussa analysoidaan TypeScriptin puutteisiin ja haittoihin liittyviä vastauksia, ja viimeisessä aliluvussa analysoidaan mahdollisia vaihtoehtoja TypeScriptille.

8.1 Kyselytutkimuksen rakenne ja suhde tutkimusongelmaan

Kyselylomake oli yksisivuinen ja kaikki paitsi yksi kysymys olivat avoimia kysymyksiä. Kyseiseen kysymykseen tuli valita vastaukseksi yksi numerovaihtoehto asteikolta 1–10. Kyselylomakkeen yläosassa oli kyselyn esittely (liite B). Kaikki kyselyn kysymykset löytyvät liitteestä C ja kyselyn saatekirje löytyy liitteestä A.

Taulukko 8.1 kuvaa kyselylomakkeen kysymysten suhdetta luvussa 6.1 esitettyihin tutkimuskysymyksiin. Taulukon avulla pystyy näkemään, minkä kysymyksen tai kysymysten avulla on pyritty saamaan vastauksia mihinkä tutkimuskysymykseen. Taulukossa kyselylomakkeen kysymykset on esitetty suomeksi, vaikka varsinainen kysely suoritettiin englanniksi. Kaikki kyselylomakkeen kysymykset, paitsi viimeinen kysymys, liittyvät tavalla tai toisella johonkin tutimuskysymykseen. Viimeistä kysymystä käytettiin ainoastaan vastausten rymittelyä varten. Taulukossa 8.1 X-arvot merkitsevät suoraa yhteyttä tutkimuskysymyksen ja kyselylomakkeen kysymyksen välillä, ja suluissa olevat X-arvot merkitsevät epäsuoraa yhteyttä näiden välillä.

	TK1: Miten hyödylliseksi ohjelmointikieleksi TypeScript koetaan?	**TK2:** Millä ohjelmistokehityksen osa-alueilla TypeScript tarjoaa eniten hyötyjä?	**TK3:** Koetaanko TypeScriptin käyttöön liittyvän haittoja?	**TK4:** Mille ohjelmistokehityksen osa-alueille TypeScriptin haitat keskittyvät?	**TK5:** Millä muulla tavalla JavaScriptiin liittyvät ongelmat ja puutteet on ratkaistu, jos TypeScript ei ole käytössä?
1. Mitkä ovat TypeScriptin hyödyllisimmät ominaisuudet, verrattuna pelkkään JavaScriptiin?	X	X			
2. Mitkä ovat TypeScriptin merkittävimmät heikkoudet / puutteet, verrattuna pelkkään JavaScriptiin?	(X)		X	X	
3. Onko TypeScript millään tavalla vaikuttanut ohjelmistokehitysprosessin suunnitteluvaiheeseen? Millä tavalla siinä tapauksessa?	(X)	X	(X)	X	
4. Millä tavalla TypeScript on vaikuttanut ohjelmistokehitysprosessin kehitysvaiheeseen?	(X)	X	(X)	X	
5. Millä tavalla TypeScript on vaikuttanut testien kirjoittamista koskeviin käytäntöihin, verrattuna käytettäessä pelkkää JavaScriptiä?	(X)	X	(X)	X	
6. Millä tavalla TypeScript on vaikuttanut ohjelmistokehitysprosessin ylläpitovaiheeseen?	(X)	X	(X)	X	
7. Oletko käyttänyt TypeScriptiä, mutta sen jälkeen luopunut sen käytöstä. Mikä oli syy luopumiselle?			(X)		X
8. Mikäli vastasit myöntävästi edelliseen kysymykseen, oletko valinnut korvaavan vaihtoehdon TypeScriptille? Mikä vaihtoehto?					X
9. Kuinka todennäköisesti suosittelisit TypeScriptiä ystäville tai kollegoille?	X		X		
10. Työnantajan nimi?					

Taulukko 8.1: Kyselylomakkeen kysymysten suhde tutkimuskysymyksiin.

8.2 TypeScriptin hyödyllisyys

Kyselylomake sisälsi sekä suorasti, että epäsuorasti TypeScriptin hyödyllisyyteen liittyviä kysymyksiä.

Ensimmäisessä kysymyksessä kysyttiin suoraan, mitkä vastaajan mielestä ovat TypeScriptin hyödyllisimmät ominaisuudet, verrattuna pelkkään JavaScriptiin. Kysymyksillä 3–6 pyrittiin epäsuoralla tavalla löytämään TypeScriptin hyödyllisyyteen liittyviä viitteitä kysymällä, millä tavalla sen käyttö on vaikuttanut ohjelmistokehitysprosessin suunnitteluvaiheeseen, -kehitysvaiheeseen, -ylläpitovaiheeseen sekä testien kirjoittamista koskeviin käytäntöihin verrattuna pelkkään JavaScriptiin. Kysymysten epäsuora yhteys TypeScriptin hyödyllisyyteen johtuu siitä, että samoilla kysymyksillä pystyttiin keräämään vastauksia myös TypeScriptin mahdollisiin puutteisiin liittyen, joten yhteys hyödyllisyyteen riippui vastauksesta ja on siksi epäsuora.

Kyselylomakkeen yhdeksännessä kysymyksessä kysyttiin, kuinka todennäköisesti vastaaja suosittelisi TypeScriptiä ystävilleen tai kollegoilleen asteikolla 1–10. Kyseisellä kysymyksellä pyrittiin saamaan yleinen arvio TypeScriptin hyödyllisyydestä laskemalla vastausten perusteella TypeScriptin NPS-pisteiden määrä. Vastaajista kolme laskettiin arvostelijoiksi, neljä passiivisiksi ja kolmetoista suosittelijoiksi. Vastaajia oli yhteensä kaksikymmentä, joten lopulliseksi NPS-pistemääräksi saatiin 50, joka merkitsee erinomaista tulosta [Sal19].

Kyselylomakkeen kysymysten vastauksia analysoimalla saatiin tunnistettua suuri määrä mielenkiintoisia havaintoja. Nimeämällä havainnot ja sen jälkeen yhdistämällä samankaltaiset havainnot yhteen saatiin lopulta tunnistettua kuusi eri TypeScriptin hyödyllisyyteen liittyvää teemaa. Seuraavaksi esitellään tunnistetut teemat ja kuvaillaan perustelut kunkin teeman taustalla. Teemojen jälkeen esitellään NPS-kyselyn tulos.

Staattinen koodianalyysi ja tyyppitarkistus

Selkeä enemmistö vastaajista oli sitä mieltä, että yksi TypeScriptin hyödyllisimmistä ominaisuuksista on kielen tarjoama staattinen koodianalyysi ja tyyppitarkistus. Osaksi tunnistettua teemaa voidaan myös laskea TypeScriptin tyyppijärjestelmän mahdollistama eri tyyppien ja tyyppiyhdistelmien kirjo sekä mahdollisuus käyttää yhdessä paikassa määriteltyä tyyppiä useassa eri paikassa ympäri koodia.

Vastauksista kävi erityisesti ilmi, että yhteiskäyttöisten tyyppien käyttö yhdessä käännösaikaisen koodianalyysin kanssa tuo mukanaan sellaisen hyödyn, että useat mahdolliset virheet saadaan tunnistettua jo käännösvaiheessa, mikä puolestaan johtaa pienempään määrään suoritusaikaisia virheitä verrattuna siihen, jos käytettäisiin pelkkää JavaScriptiä. Tyyppien yhteisen käytön hyötynä mainittiin se, että jos tyyppiä muutetaan yhdessä kohtaa, osaa koodianalyysi ilmoittaa virheestä, jos kyseistä tyyppiä käytetään väärällä tavalla jossain toisessa kohdassa.

Koodianalyysin hyvänä ominaisuutena mainittiin lisäksi, että sen avulla saadaan kiinni myös erinäiset huolimattomuusvirheet, kuten esimerkiksi muuttujan nimen kirjoitusvirhe, ennen kuin ne saatetaan osaksi tuotannossa suoritettavaa koodia.

Refaktoroinnin helppous

Suuri osa vastaajista oli myös sitä mieltä, että TypeScript-koodia on helpompi refaktoroida kuin pelkkää JavaScript-koodia. Refaktorointi tarkoittaa prosessia, jossa tietokoneohjelman lähdekoodia muutetaan siten, että ohjelman toiminnallisuus säilyy, mutta koodin rakenne muuttuu. Muutokset voivat koskea esimerkiksi koodin luettavuutta tai ohjelmakomponenttien työnjaon selkeyttämistä.

Vastausten mukaan refaktorointiteema on kytköksissä myös edellä mainittuun *Staattinen koodianalyysi ja tyyppitarkistus* -teemaan, koska sen avulla kehittäjä saa heti varoituksia siitä, jos muutettu koodi on aiheuttanut virheitä muualle koodiin, ja sen myötä vältytään mahdollisilta regressiovirheiltä. Regressiolla tarkoitetaan sitä, kun olemassa olevaan toimivaan koodiin ilmaantuu virheitä muutettaessa ohjelmankoodin toista osaa.

Vastaajien mukaan refaktoroinnin helppous vaikuttaa suuresti myös koodin laatuun, koska koodin laatua parantavaa refaktorointia uskaltaa ylipäätänsä tehdä laajemmassa mittakaavassa, kun voi luottaa koodianalyysin ilmoittavan mahdollisista syntyvistä virheistä.

Alla lainaus yhdeltä vastaajalta, jonka mielestä staattinen tyyppitarkistus auttaa huomaamaan mahdolliset virheet ja lisää luotettavuutta muutosten tekoon refaktoroinnin yhteydessä.

> *"Static type checking helps to notice possible places for bugs and gives confidence to changes when doing refactoring."*

Kooditason dokumentaatio

TypeScriptin tyyppimerkinnät voivat toimia eräänlaisena kooditason dokumentaationa, koska tyyppimerkintöjen perusteella pystyy esimerkiksi näkemään, minkä tyyppisiä arvoja kukin funktio vastaanottaa parametreina ja minkä tyyppisiä arvoja funktiot palauttavat. Tyyppimerkintöihin sidotun dokumentaation hyvä puoli on myös se, että dokumentaatio ei pääse vanhenemaan vaan pysyy aina ajan tasalla eikä dokumentaatiota tarvitse erikseen etsiä mistään toisesta paikasta.

Useat vastaajista olivat myös sitä mieltä, että tyyppimerkintöjen ansiosta olemassa olevan koodin ymmärtäminen helpottuu, jolloin ei ole ainoastaan suoraviivaisempaa palata aikaisempaan koodiin ja ymmärtää nopeasti, mitä

koodin on tarkoitus tehdä, vaan uusien kehittäjien perehdyttäminen on myös helpompaa, kun koodia ei tarvitse käydä yhtä yksityiskohtaisesti läpi vaan se selittää itse itsensä.

Automaattitäydennys- ja koodivihjeet

TypeScriptin tarjoaman muodollisen tyyppitiedon ansiosta kehitystyökalujen on mahdollista tarjota kehittäjälle täsmällisiä asiayhteydellisiä automaattitäydennysvihjeitä. Noin puolet kyselyn vastaajista ilmaisivat TypeScriptin merkittävimpiin hyötyihin kuuluvan sen mahdollistamat edistykselliset kehitystyökalujen automaattitäydennysvihjeet. TypeScriptin tyyppimerkintöjen ja tyyppipäättelyn ansiosta kehitystyökalut tietävät tarkalleen, minkälaisia arvoja kehittäjä on käsittelemässä, ja osaa tarjota juuri kyseisiin arvoihin liittyviä automaattitäydennysvihjeitä. Vastaajien mukaan vihjeet vähentävät esimerkiksi lähdekoodissa edestakaisin hyppimistä ja nopeuttavat sen myötä kehitystyötä.

Tarve vähemmille testeille

Kyselylomakkeen vastaukset osoittavat vastaajien olevan sitä mieltä, että TypeScript-ohjelmiin ei tarvitse kirjoittaa yhtä paljon testejä kuin pelkällä JavaScriptillä tehtyihin ohjelmiin. Koska TypeScriptin käännösaikainen koodianalyysi pitää huolen siitä, että oikeantyyppisiä arvoja käytetään oikeissa paikoissa, ei tyyppeihin liittyviä testejä käytännössä tarvitse luoda lainkaan. Tyyppeihin liittyvillä testeillä tarkoitetaan testejä, joiden avulla varmistetaan ohjelman oikea käyttäytyminen myös odottamattoman tyyppisillä arvoilla.

Vastaajien mielestä vaikutus ei koske ainoastaan tyyppeihin liittyviä testejä, vaan että TypeScriptin myötä yksikkötestien kokonaismäärän tarve on myös pienempi verrattuna pelkällä JavaScriptillä toteutettuihin ohjelmiin.

Uusimmat ECMAScript-ominaisuudet käyttöön

Osa vastaajista mainitsi yhtenä TypeScriptin merkittävimmistä hyödyistä sen, että TypeScriptin avulla uusimmat ECMAScript ominaisuudet on helppo ottaa käyttöön hyvin aikaisessa vaiheessa, koska TypeScript-koodin perusteella generoitavan JavaScript-koodin ECMAScript-version pystyy itse määrittelemään.

8.3 TypeScriptin hyödyllisyys ohjelmistokehityksen eri osa-alueilla

Kyselylomakkeen kysymyksillä 3–6 pyrittiin saamaan muodostettua kuva siitä, miten TypeScript on vaikuttanut vastaajien mielestä heidän ohjelmistokehitysprosessinsa suunnittelu-, kehitys- ja ylläpitovaiheisiin sekä yleisesti heidän koko testausprosessiinsa. Seuraavaksi esitellään vastaajien mieltämät TypeScriptin hyödyt jokaisen edellä mainitun jaottelun osalta.

Suunnitteluvaihe

Ohjelmistokehitysprosessin suunnitteluvaiheen osalta vastaajat olivat sitä mieltä, että suurin TypeScriptin mukanaan tuoma hyödyllisyys on mahdollisuus suunnitella ja hahmottaa ohjelmassa käytettävien tyyppien rakennetta, ennen kuin kirjoitetaan riviäkään koodia. Vastaajat olivat sitä mieltä, että panostamalla tyyppien suunniteluun koko ohjelman rakenteesta saadaan selkeämpi ja erilaisten ohjelmassa välitettävien tietorakenteiden määrää saadaan vähennettyä, mikä puolestaan parantaa ohjelman kokonaislaatua.

Osa vastaajista nosti esille myös sellaisen suunnitteluun liittyvän näkökulman, että he ovat TypeScriptin käytön myötä alkaneet selkeästi suosia sellaisten kolmannen osapuolen ohjelmakirjastojen käyttöä, jotka ovat tehty TypeScriptillä, koska silloin kirjaston tyyppimerkinnät ovat myös automaat-

tisesti ajan tasalla.

Kehitysvaihe

Osa vastaajista oli sitä mieltä, että TypeScriptillä on ollut sellainen vaikutus ohjelmointiprosessin kehitysvaiheeseen, että koodin tuottaminen on tehokkaampaa ja lopputuloksena on yksinkertaisempaa koodia verrattuna siihen, jos olisi käytetty pelkkää JavaScriptiä. Osa vastaajista oli tosin sitä mieltä, että uusien ominaisuuksien luominen saattaa olla TypeScriptillä hitaampaa, mutta olemassa olevien toiminnallisuuksien muuttaminen on paljon turvallisempaa ja nopeampaa. Syyksi moni vastaajista painotti sitä, että TypeScriptin tyyppien toimiessa koodiin kytketyn dokumentaation tavoin on helpompi ymmärtää olemassa olevaa koodia eikä uusia ominaisuuksia myöskään tarvitse dokumentoida erikseen yhtä kattavasti.

Vastaajat olivat lisäksi sitä mieltä, että TypeScriptin tyypit ja kääntäjän ilmoittamat mahdolliset koodianalyysivirheet auttavat ohjaamaan kehitysprosessia ja helpottavat löytämään ne kohdat koodista, jotka edelleen kaipaavat työstöä.

Testausprosessi

Vastaajien mielestä suurin vaikutus, mikä TypeScriptillä on ollut heidän testausprosesseihinsa, on se, että tarvittavien yksikkötestien määrä on laskenut huomattavasti ja tyyppeihin liittyvistä testeistä on käytännössä voitu luopua kokonaan. Vastaajat kertoivat myös, että koska testeissä käytettäville arvoille voidaan määrittää tyypit, pitää käännösaikainen koodianalyysi huolen siitä, että jos varsinaisen toteutuksen tyyppejä päivitetään, tulee samalla päivitettyä myös testejä, tai muuten koodianalyysi ilmoittaisi virheestä. Esille nostettu hyöty on, että testit pysyvät ajan tasalla ja niitä on nopea päivittää.

Ylläpitovaihe

Useat vastaajista olivat sitä mieltä, että TypeScriptillä on ollut ohjelmistojen ylläpitoa helpottava vaikutus, ja mielipiteen taustalla on monia syitä. Yksi syistä on, että koska käännösaikainen koodianalyysi osaa jo kehitysvaiheessa varoittaa monista virheistä, ei tuotantoon asti pääse yhtä paljon virheitä ja sillä on helpottava vaikutus ohjelmiston ylläpitoon. Toinen mainittu syy on, että kuten luvussa 8.2 todettiin, on olemassa olevan TypeScript-koodin refaktorointi ja muutosten tekeminen siihen paljon luotettavampaa ja helpompaa verrattuna pelkkään JavaScript-koodiin. Kolmantena ja viimeisenä syynä mainittiin se, että koska tyypit toimivat kooditason dokumentaationa, on olemassa olevan koodin ymmärtäminen paljon helpompaa ja virheiden paikantaminen sellaisesta koodista, jota ymmärtää, on paljon helpompaa kuin sellaisesta koodista, jota ei ymmärrä.

8.4 TypeScriptin puutteet ja haitat

Kuten TypeScriptin hyödyllisyyteen liittyen sisälsi kyselylomake myös sekä suorasti että epäsuorasti TypeScriptin puutteisiin ja haittoihin liittyviä kysymyksiä.

Toisessa kysymyksessä kysyttiin suoraan, mitkä vastaajan mielestä ovat TypeScriptin merkittävimmät heikkoudet tai puutteet verrattuna pelkkään JavaScriptiin. Kysymyksillä 3–7 pyrittiin epäsuoralla tavalla löytämään TypeScriptiin liittyviä negatiivisia mielipiteitä.

Vastausten perusteella saatiin tunnistettua viisi TypeScriptin puutteisiin tai haittoihin liittyvää teemaa. Kyseisiä teemoja ei niiden yleisen luonteen takia saatu sidottua mihinkään tiettyyn ohjelmistokehitysprosessin vaiheeseen. Seuraavaksi esitellään tunnistetut teemat sekä perustelut niiden taustalla.

Puutteelliset tyyppimäärittelyt

Kyselylomakkeen vastausten perusteella selkeästi merkittävämmäksi TypeScriptiin liittyväksi puutteeksi tunnistettiin kolmannen osapuolen ohjelmakirjastojen puutteelliset tai olemattomat tyyppimäärittelyt.

Puutteelliset tyyppimäärittelyt voivat johtua siitä, että ohjelmakirjastoa ei ole toteutettu TypeScriptin avulla vaan tyyppimäärittelyt on lisätty jälkikäteen sellaisessa muodossa, että ne eivät vastaa ohjelmakirjaston toteutusta. Puutteelliset tyyppimäärittelyt aiheuttavat päänvaivaa, kun ohjelmakirjastoa käytetään tyyppimäärittelyn mukaisesti mutta määrittely ei vastaakaan ohjelmakirjaston todellista toteutusta. Silloin ohjelmaan saattaa lopputuloksena syntyä suoritusaikaisia virheitä.

Kokonaan puuttuvien tyyppimäärittelyjen osalta kehittäjällä ei ole käytössään kooditason dokumentaatiota tyyppimerkintöjen muodossa, eivätkä kehitystyökalut osaa tarjota yhtä tarkkoja automaattitäydennysvirheitä, kuin jos tyyppimäärittely olisi saatavilla. Tyyppimäärittelyjä pystyy luomaan itse, mutta on olemassa riski, että itse tehty tyyppimäärittely ei vastaakaan kaikilta osin ohjelmakirjaston toteutusta, jolloin vastassa ovat taas edellä kuvatut haasteet.

Tyyppijärjestelmän puutteet

Kyselylomakkeen vastauksista ilmeni myös, että moni vastaajista koki TypeScript-tyyppijärjestelmän puutteiden olevan osana TypeScriptin merkittävimpiä puutteita. Kaksi asiaa nostettiin erityisesti esille: vaikeasti tulkittavat tyyppivirheet ja ajoittain puutteellinen tyyppipäättely.

Vaikeasti tulkittavat tyyppivirheet koettiin ongelmallisiksi, koska virheiden korjaaminen koettiin haastavaksi, kun on epäselvää, minkälaisesta virheestä on kyse. Ajoittain puutteellinen tyyppipäättely aiheutti puoles-

taan turhautumista, koska toimivaan koodiin on välillä lisättävä ylimääräisiä tyyppimerkintöjä tai ylimääräistä koodia, jotta kääntäjä ymmärtäisi tavan, jolla tyyppejä on käytetty.

Tarve kääntää TypeScript-koodi JavaScript-koodiksi

Osa vastaajista oli sitä mieltä, että yksi TypeScriptin puutteista tai haitoista on se, että TypeScript-koodi on käännettävä JavaScript-koodiksi, ennen kuin sitä pystyy suorittamaan esimerkiksi selaimessa.

Ongelmalliseksi koettiin muun muassa se, että koodia ei pysty suorittamaan sellaisenaan, jolloin sen toimivuutta ei esimerkiksi voida suoraan kokeilla eri selainversioilla vaan koodi on ensin käännettävä JavaScript-koodiksi. Perusteluissa mainittiin myös, että TypeScript-kääntäjän vaatimat asetukset vievät aikaa, joka saattaa erityisesti pienten projektien tai prototyyppiohjelmien tapauksessa jopa hidastaa kehitystyötä.

Virheellinen turvallisuuden tunne

Kaksi vastaajista oli sitä mieltä, että TypeScriptin tarjoama staattinen koodianalyysi saattaa antaa virheellisen turvallisuuden tunteen, jos luotetaan siihen, että jos käännösaikainen anlyysi ei havaitse virheitä, ei niitä voi myöskään syntyä suoritusaikaisesti. Tosiasiassa, esimerkiksi rajapintojen kautta tulevat ulkoiset tarkistamattomat syötteet tai `any`-tyypin avulla tyypitetyt arvot saattavat olla suoritusaikaisesti erityyppisiä kuin on oletettu, ja se voi johtaa suoritusaikaisiin poikkeuksiin ohjelmassa.

Vaatii opettelua

Kaksi vastaajista oli sitä mieltä, että pieni puute TypeScript-kielen käytön suhteen on se tosiasia, että vaikka kieli on pohjimmiltaan JavaScriptiä,

vaatii sen opettelu joka tapauksessa jonkin verran aikaa. TypeScriptin tyyppijärjestelmän täyden potentiaalin hyödyntäminen vaatii vastaajien mielestä erityisesti opettelua, jos oletetaan, että kehittäjä on aikaisemmin ainoastaan käyttänyt JavaScript-kieltä.

8.5 Vaihtoehdot TypeScriptille

Kyselylomakkeen kahdeksannessa kysymyksessä kysyttiin, mikäli TypeScriptin käytöstä on luovuttu, niin mitä korvaavaa vaihtoehtoa on käytetty sen tilalla. Kysymyksen tavoitteena oli saada kuva mahdollisista varteenotettavista ohjelmointikielivaihtoehdoista TypeScriptille. Vain kaksi vastaajaa oli jossain vaiheessa luopunut TypeScriptin käytöstä, mutta vastauksia mahdollisista vaihtoehdoista saatiin silti neljä kappaletta. Yksi mainituista vaihtoehdoista oli käyttää pelkkää JavaScriptiä ja kolme muuta olivat Flow [Fac20], Elm [ELM20] sekä Flutter [Goo20]. Valitettavasti vastauksista ei ilmennyt juuri mitään perusteluja sille, miksi juuri kyseiset ohjelmistokielet olisivat varteenotettavia vaihtoehtoja TypeScriptille.

9 Johtopäätökset

Tutkielmassa suoritetun kyselytutkimuksen avulla tutkimusongelmaan saatiin vastauksia ja tutkimuksen tavoitteisiin päästiin, sillä vastaukset vahvistivat ikaisemmissa tieteellisissä tutkimuksissa tehtyjä havaintoja. TypeScriptin hyötyjen osalta saatiin myös selvyys siitä, millä ohjelmistokehitysprosessin osa-alueilla ne korostuvat.

TypeScriptin puutteiden ja haittojen osalta saadut vastaukset jäivät valitettavan tyngiksi, eikä niiden osalta saatu niin kattavia tutkimustuloksia, kuin olisi toivottu.

Tässä luvussa käydään läpi, millä tavalla tutkimustulokset vertautuvat aikaisempaan tutkimukseen, minkälaiset vastaukset tutkimuskysymyksiin saatiin, tutkimustulosten validiteettiuhat sekä mahdollisia jatkotutkimusaiheita.

9.1 Tutkimustulokset ja aikaisempi tutkimus

Kuten luvussa 5.1 todettiin, tutkivat Meijer ja Drayton vuonna 2005, olisiko staattinen vai dynaaminen tyypitys parempi lähtökohta ohjelmointikielelle [MD04]. Tutkimuksen lopputulokseksi saatiin, että staattinen tyypitys on tehokas työkalu, joka auttaa kehittäjiä ilmaisemaan olettamuksia ongelmasta, jota he yrittävät ratkaista, ja antaa heille mahdollisuuden kirjoittaa ytimekkäämpää ja virheettömämpää koodia. Kyseisen tutkimuksen tulokset vastaavat täysin tämän tutkimuksen tuloksia, koska vastaajat olivat sitä mieltä, että stattisesti tyypitetyn TypeScriptin avulla voidaan kirjoittaa luotettavampaa ja virheettömämpää koodia verrattuna dynaamisesti tyypitettyyn JavaScriptiin. Meijer ja Drayton totesivat kumminkin myös, että dynaamista tyypitystä tulisi voida käyttää silloin, kun se on tarpeellista. TypeScriptissä dynaaminen tyypitys on tarvittaessa mahdollista `any`-tyypin

avulla.

Kuten luvussa 5.3 todettiin, suorittivat Gao, Bird & Barr vuonna 2017 tutkimuksen [GBB17], jossa he yrittivät selvittää, minkälaisia hyötyjä staattisten tyyppijärjestelmien avulla voisi saavuttaa JavaScript-kehityksessä. Toinen tutkittavista tyyppijärjestelmistä oli TypeScriptin versio 2.0. Tutkimuksen lopputulokseksi saatiin, että TypeScriptin avulla 15 % tutkituista virheistä olisi voitu havaita jo ohjelmien kehitysvaiheessa staattisen koodianalyysin ansiosta. Kyseiset tulokset ovat täysin linjassa tämän tutkimuksen tulosten kanssa, koska kyselyvastausten perusteella juuri staattinen koodianalyysi miellettiin yhdeksi TypeScriptin hyödyllisimmistä ominaisuuksista, jonka avulla voidaan havaita mahdollisia virheitä jo kehitysvaiheessa, ja näin ollen estää niiden joutuminen tuotannossa suoritettavaan koodiin.

Luvussa 5.2 esitellyssä tutkimuksessa, jonka Ocariza Jr., Bajaj, Pattabiraman & Mesbah suorittivat vuonna 2017, tutkittiin JavaScript-ohjelmointivirheiden syitä ja seuraamuksia [OBPM17]. Suuri osa tutkimuksessa havaituista ohjelmointivirheistä olisi mahdollisesti voitu välttää TypeScriptin staattisen koodianalyysin ansiosta, koska 33 % virheistä oli tyypityksiin liittyviä ja peräti 75 % kaikista virheistä johtuivat siitä, että JavaScript-metodia oli kutsuttu odottamattomalla tai virheellisellä arvolla.

9.2 Tutkimuskysymyksiin vastaaminen

Tutkielma vastaa tutkimuskysymyksiin seuraavasti:

TK1: *Miten hyödylliseksi ohjelmointikieleksi TypeScript koetaan?*
TypeScript koetaan erittäin hyödylliseksi ohjelmointikieleksi ja sen käyttöön koetaan liittyvän monia erilaisia hyötyjä. TypeScriptin NPS-pistemääräksi saatiin 50, joka on erinomainen tulos. TypeScriptin avulla voidaan ratkaista

useita JavaScriptiin liittyviä tunnistettuja puutteita, ja lisäksi TypeScript tuo mukanaan hyötyjä muun muassa staattisen koodianalyysin ja tyyppitarkistuksen sekä kooditason dokumentaationa toimivien tyyppimerkintöjen muodossa. Käännösaikainen koodianalyysi tekee koodin refaktoroinnista paljon helpompaa ja luotettavampaa, ja kääntäjän varoittaessa virheistä jo kehitysvaiheessa on lopputuloksena vähemmän suoritusaikaisia virheitä. Staattisen koodianalyysin ansiosta voidaan myös vähentää yksikkötestien määrää, ja tyyppejä koskevista testeistä voidaan mahdollisesti jopa kokonaan luopua.

TK2: *Millä ohjelmistokehityksen osa-alueilla TypeScript tarjoaa eniten hyötyjä?*

TypeScriptin koetaan tarjoavan hyötyjä monilla eri ohjelmistokehityksen osa-alueilla. Suunnitteluvaiheessa voidaan suunnitella tyyppien rakenne, joka auttaa hahmottamaan ohjelmakokonaisuuden tarpeita. Kehitysvaiheessa saadaan apua kehitystyökalujen kattavista automaattitäydennysvihjeistä, ja tyyppimerkintöjen toimiessa dokumentaationa on helpompi ymmärtää olemassa olevaa koodia. Staattinen koodianalyysi auttaa myös havaitsemaan mahdollisia virheitä jo kehitysvaiheessa. Testausvaiheen hyödyt painoittuvat siihen, että yksikkötestien määrää voidaan vähentää ja tyyppejä testaavista testeistä voidaan mahdollisesti jopa kokonaan luopua. Ylläpitovaiheen hyötyihin kuului, että kooditason dokumentaatio auttaa ymmärtämään olemassa olevaa koodia ja siihen palaaminen on helpompaa, ja koodin refaktorointi on paljon helpompaa ja luotettavampaa, koska staattinen koodianalyysi toimii apuna regressiovirheiden välttämisessä.

TK3: *Koetaanko TypeScriptin käyttöön liittyvän haittoja?*

TypeScriptin käyttöön koetaan liittyvän myös joitain haittoja. Yksi esimerkki on, että pienten projektien ja prototyyppien osalta TypeScript saattaa jopa toimia hidastavana tekijänä, koska TypeScript koodia ei voi ajaa sellaisenaan selaimessa, joten kääntäjän asetusten asettamiseen kuluu ylimääräistä aikaa. Toinen koettu haitta on, että TypeScriptin tyyppijärjestelmän opetteluun kuluu aikaa, jos siitä ei ole aikaisempaa kokemusta.

Havaittujen haittojen lisäksi TypeScript-kielessä tai sen käyttöön liittyen koetaan olevan muutamia puutteita. Merkittävimmät puutteet ovat, että tyyppijärjestelmän tyyppipäättely ei aina täysin ymmärrä luotuja tyyppejä ja että kolmannen osapuolen ohjelmakirjastojen tyyppimäärittelyt saattavat olla puutteellisia tai puuttua kokonaan.

TK4: *Mille ohjelmistokehityksen osa-alueille TypeScriptin haitat keskittyvät?*

Tutkimuksen tulosten perusteella ei pystytty tunnistamaan mitään tiettyjä ohjelmistokehityksen osa-alueita, joille TypeScriptin haitat tai puutteet keskittyisivät.

TK5: *Millä muulla tavalla JavaScriptiin liittyvät ongelmat ja puutteet on ratkaistu, jos TypeScript ei ole käytössä?*

Tutkimuksen tulosten perusteella ei pystytä esittämään mitään varteenotettavaa vaihtoehtoa TypeScriptille, joka ratkaisisi JavaScriptiin liittyvät ongelmat ja puutteet. Kaksi kyselyyn vastanneista oli jossain vaiheessa luopunut TypeScriptin käytöstä, mutta hekin olivat palanneet käyttämään pelkkää JavaScriptiä, joten samat ongelmat ja puutteet ovat yhä läsnä. Muut vastaajaat suosittelivat vaihdoehdoiksi kolmea muuta ohjelmointikieltä, mutta tarkempien perustelujen puuttuessa vaatii aihe lisätutkimusta, ennen kuin

voidaan tehdä tarkempia johtopäätöksiä.

9.3 Validiteettiuhat

Tutkimukseen osallistuneiden vastaajien alhainen määrä on ensimmäinen tunnistettu validiteettiuhka. Vaikka vastauksia saatiin tutkielman luonteeseen ja kyselyn vastaajille asetettuihin soveltuvuusvaatimuksiin nähden ihan kohtuullinen määrä, jäätiin vastaajien määrässä selkeästi vajaaksi tavoitteeksi asetetusta määrästä 30–500. Vastauksia tarvittaisiin huomattavasti enemmän, jotta tulosten perusteella voitaisiin tehdä tilastollisesti merkittäviä johtopäätöksiä.

Toinen mahdollinen validiteettiuhka on se, että ovatko vastaajat osanneet tulkita kyselyn kysymykset oikein vai onko vastauksissa oikeasti vastattu eri asioihin kuin mitä on kysytty. Ovatko vastaajat esimerkiksi ymmärtäneet, että kysymyksiin on toivottu vastauksia vastaajien omien kokemuksien pohjalta eikä yleisten vallitsevien käsitysten pohjalta. Vastausten perusteella pystyy päättelemään, että noin 70 % vastaajista on selkeästi vastannut omien kokemustensa pohjalta, mutta jäljelle jäävän 30 %:n osalta jää epävarmaksi jos he ovat ymmärtäneet kysymykset oikein. Jälkikäteen arvioituna, olisi voinut olla viisasta painottaa kyselylomakkeen kysymyksissä että kysymyksillä haetaan takaa nimenomaan vastaajien omia mielipiteitä.

Kolmas validiteettiuhka on, että onko saatuja vastauksia osattu tulkita oikein vai ovatko esimerkiksi omat ennakkoluuloni vaikuttaneet vastaustulosten käsittelyyn ja tulkintaan. Myös kokemattomuus tutkielmassa käytetystä temaattisen synteesin metodologiasta voi johtaa siihen, että tuloksia ei ole tulkittu oikealla tavalla.

9.4 Jatkotutkimus

Mielestäni hyvä jatkotutkimuskohde olisi tutkia, voiko TypeScriptin avulla todella vähentää yksikkötestien määrää ja onko sillä vaikutusta ohjelmiston laatuun. Olisi myös mielenkiintoista saada tarkempaa tietoa siitä, miten paljon kattavampia tai tarkempia automaattitäydennys- ja koodivihjeitä kehitystyökalut voivat tarjota TypeScriptin avulla verrattuna pelkkään JavaScriptiin. Lopuksi, jotta viidenteen tutkimuskysymykseen saataisiin selkeä vastaus, vaatisi sen aihe lisätutkimusta eli onko TypeScriptille olemassa muita varteenotettavia vaihtoehtoja, joiden avulla JavaScriptiin liittyvät ongelmat ja puutteet saataisiin ratkaistua, ja mikä olisi vaihtoehdoista paras.

Lähteet

[Ast15] Aston, Ben: *A brief history of JavaScript*, 2015. `https://medium.com/@benastontweet/lesson-1a-the-history-of-javascript-8c1ce3bffb17`, vierailtu 4.11.2018 .

[BAB19] BABEL: *What Is Babel?*, 2019. `https://babeljs.io/docs`, vierailtu 8.7.2019 .

[Bal99] Ball, Thomas: *The Concept of Dynamic Analysis.* ESEC/FSE'99, LNCS 1687. Springer Verlag Heidelberg, sivut 216-234, 1999.

[Bas19] Basarat, Ali Syed: *TypeScript Deep Dive - Why TypeScript*, 2019. `https://basarat.gitbooks.io/typescript/docs/why-typescript.html`, vierailtu 28.09.2019 .

[Car04] Cardelli, Luca: *Type Systems.* CRC Handbook of Computer Science and Engineering. CRC Press, 2. painos, 2004, ISBN 978-1-584-88360-9.

[CD11] Cruzes, Daniela S. ja Dybå, Tore: *Recommended Steps for Thematic Synthesis in Software Engineering.* IEEE International Symposium on Empirical Software Engineering and Measurement, sivut 275-284, 2011. `http://dx.doi.org/10.1109/ESEM.2011.36`.

[Cro08] Crockford, Douglas: *JavaScript: The good parts.* O'Reilly Media Inc., 2008, ISBN 978-0-596-51774-8.

[Dar19] Dart: *Language Tour*, 2019. `https://dart.dev/guides/language/language-tour`, vierailtu 8.7.2019 .

[ECM18] ECMA-262: *ECMAScript® 2018 Language Specification.* ECMA International, 9. painos, 2018.

[ELM20] ELM: *A delightful language with no runtime exceptions.*, 2020. `https://elm-lang.org/`, vierailtu 20.1.2020 .

[Fac20] Facebook: *Flow*, 2020. `https://flow.org/`, vierailtu 20.1.2020 .

[Fen18] Fenton, Steve: *Pro Typescript - Application scale JavaScript development.* Apress Media LLC, 2. painos, 2018, ISBN 978-1-4842-3248-4.

[Fla11] Flanagan, David: *JavaScript: The Definitive Guide.* O'Reilly Media Inc., 6. painos, 2011, ISBN 978-0-596-80552-4.

[GBB17] Gao, Zheng, Bird, Christian ja Barr, Earl T.: *To Type or Not to Type: Quantifying Detectable Bugs in JavaScript.* ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, 2017.

[Goo20] Google: *Flutter*, 2020. `https://flutter.dev/`, vierailtu 20.1.2020 .

[Kum17] Kumara, M., Sysart Oy: *Vaihtamalla paranee – 3 hyvää syytä valita TypeScript*, 2017. `https://sysart.fi/blog/2017/12/04/vaihtamalla-paranee-3-hyvaa-syyta-valita-typescript/`, vierailtu 31.8.2019 .

[MD04] Meijer, Eric ja Drayton, Peter: *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages.* OOPSLA, lokakuu 2004.

[Mic19] Microsoft: *Typescriptin virallinen dokumentaatio*, 2019. https://www.typescriptlang.org/, vierailtu 30.3.2019 .

[Moz18] Mozilla Developer Network: *What is JavaScript? - A high-level definition*, 2018. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript, vierailtu 3.10.2018 .

[Moz19a] Mozilla Developer Network: *Inheritance and the prototype chain*, 2019. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain, vierailtu 7.5.2019 .

[Moz19b] Mozilla Developer Network: *JavaScript and the ECMAScript specification*, 2019. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction#JavaScript_and_the_ECMAScript_Specification, vierailtu 2.5.2019 .

[Moz19c] Mozilla Developer Network: *Lexical Grammar - Automatic Semicolon Insertion*, 2019. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Automatic_semicolon_insertion, vierailtu 9.7.2019 .

[OBPM13] Ocariza Jr., F.S., Bajaj, K., Pattabiraman, K. ja Mesbah, A.: *An Empirical Study of Client-Side JavaScript Bugs.* ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, sivut 56-64, 2013.

[OBPM17] Ocariza Jr., F.S., Bajaj, K., Pattabiraman, K. ja Mesbah, A.: *A Study of Causes and Consequences of Client-Side JavaScript*

Bugs. IEEE Transactions on Software Engineering, Vol. 43, No. 2, February 2017, 2017.

[Pen14] Penttilä, Elias: *Master's Thesis: Improving C++ Software Quality with Static Code Analysis*. Aalto University, School of Science, 2014.

[Pey17] Peyrott, Sebastián: *A Brief History of JavaScript*, 2017. `https://auth0.com/blog/a-brief-history-of-javascript/`, vierailtu 4.11.2018 .

[Pie02] Pierce, Benjamin Crawford: *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002, ISBN 0-262-16209-1.

[Rub18] Rubens, Arden: *The History of JavaScript [INFOGRAPHIC]*, 2018. `https://www.checkmarx.com/blog/javascript-history-infographic/`, vierailtu 4.11.2018 .

[Sal19] Sales Communications: *Mikä on NPS ja miksi sen tulisi kiinnostaa yritystä?*, 2019. `https://www.salescommunications.fi/blog/mika-on-nps-ja-miksi-sen-tulisi-kiinnostaa-yritysta`, vierailtu 23.10.2019 .

[SR07] Sue, Valerie M. ja Ritter, Lois A.: *SAGE Research Methods, Conducting Online Surveys*. Sage Publications Ltd., 2007. `http://dx.doi.org/10.4135/9781412983754`.

[SSS08] Shull, Forrest, Singer, Janice ja Sjøberg, Dag: *Guide to Advanced Empirical Software Engineering*. Springer Science+Business

Media, 2008. http://link.springer.com/book/10.1007/978-1-84800-044-5.

[Sta18] Stack Overflow, Stack Exchange Inc.: *Developer Survey Results 2018*, 2018. https://insights.stackoverflow.com/survey/2018, vierailtu 28.1.2019 .

[Sta19] Stack Overflow, Stack Exchange Inc.: *Developer Survey Results 2019*, 2019. https://insights.stackoverflow.com/survey/2019#most-popular-technologies, vierailtu 13.4.2019 .

[Tar16] Tarvainen, J., Symfony Finland: *Mikä on Typescript*, 2016. https://symfony.fi/artikkeli/mika-on-typescript, vierailtu 28.1.2019 .

[VJ06] Van Selm, Martine ja Jankowski, Nicholas W.: *Quality & Quantity 40: 435-456, Conducting Online Surveys*. Kluwer Academic Publishers, 2006. https://doi.org/10.1007/s11135-005-8081-8.

A Sahköpostitse toimitettava saatekirje

Dear software industry representative,

My name is Jani Rapo and I'm a student at the Department of Computer Science at the University of Helsinki. I'm doing my master's thesis on the subject **The usefulness of TypeScript in JavaScript programming**, and as a part of my thesis I'm conducting an online survey to gain insight on how useful or harmful TypeScript is considered to be among experts working in the software development industry.

You have received this message because you are working for a company that practices software development and are either yourself a suitable respondent to the survey or you are able to forward this message to other possible respondents at your company. The most suitable respondents are people who are either currently working with software development or have worked with it in the past, and who have hands-on experience with both JavaScript and TypeScript programming.

Whether you are a suitable respondent or not, I would in any case kindly wish you to convey this message to potential respondent candidates in your company, in order to obtain a comprehensive sample of expert opinions as a part of the research for my thesis.

The survey contains ten questions and takes about 15 minutes to complete. The survey is answered anonymously and the identity of the respondent cannot be traced back to the answers. It is possible to give the name of the company you are representing as an optional answer in the survey, but that information is not published in any form as a part of the thesis, and is only used to possibly group results together during the analysis of the results.

The survey will be available during the time period 19.11.2019 – 4.12.2019 from the following link *[kyselyn URL]* and the final thesis will be published during the first half of the year 2020.

Many thanks in advance!

— — — — — — — — — — — — — — — —

Hei arvoisa ohjelmistoalan edustaja!

Olen Jani Rapo Helsingin yliopiston tietojenkäsittelytieteen laitokselta ja teen pro gradu -tutkielmaani aiheesta **TypeScriptin hyödyllisyys JavaScript-ohjelmistokehityksessä**. Tutkielmani osana on kysely, jonka tarkoituksena on saada näkemystä siihen, kuinka hyödyllisenä tai haitallisena TypeScriptiä pidetään web-ohjelmistoteollisuuden yrityksissä työskentelevien asiantuntijoiden keskuudessa.

Te olette saaneet tämän viestin, koska työskentelette ohjelmistokehitystä harjoittavassa yrityksessä ja olette joko itse soveltuva vastaajaksi, tai pystytte välittämään kyselyn eteenpäin muille soveltuville vastaajille yrityksessänne. Kyselyyn vastaajiksi soveltuvat parhaiten henkilöt, jotka ovat joko aikaisemmin työskennelleet tai työskentelevät parhaillaan ohjelmistokehityksen parissa ja joilla on omakantaista kokemusta sekä JavaScript- että TypeScript-ohjelmoinnista.

Olitte sitten soveltuva vastaajaksi tai ette, toivoisin, että joka tapauksessa välittäisitte tämän viestin eteenpäin mahdollisille vastaajaehdokkaille yrityksessänne, jotta saisin tutkielmaani kattavan otannan asiantuntijoiden mielipiteitä.

Kyselyyn vastaaminen tapahtuu nimettömasti, eikä vastaajan henkilöllisyyt-

tä pystytä jäljittämään vastausten perusteella. Kyselyssä on mahdollista kertoa valinnaisena tietona yrityksen nimi, jossa työskentelee, mutta sitäkään tietoa ei julkaista missään muodossa, vaan tietoa käytetään ainoastaan vastausten ryhmittelyyn tulosten analyysivaiheessa.

Kysely sisältää kymmenen kysymystä ja kyselyyn vastaamiseen kuluu aikaa noin 15 minuuttia. Kyselyn kysymykset on esitetty englanniksi, mutta vastaaminen on mahdollista joko suomeksi tai englanniksi. Kyselyyn on mahdollista vastata 19.11.2019 – 4.12.2019 välisenä aikana seuraavan linkin kautta: *[kyselyn URL]*. Lopullinen tutkielma julkaistaan vuoden 2020 alkupuoliskolla.

Suuri kiitos jo etukäteen!

B Kyselyn etusivu ja esittely

My name is Jani Rapo and I'm a student at the Department of Computer Science at the University of Helsinki. I'm doing my master's thesis on the subject **The usefulness of TypeScript in JavaScript programming**, and as a part of my thesis I'm conducting an online survey to gain insight on how useful or harmful TypeScript is considered to be among experts working in the software development industry.

The most suitable respondents are people who are either currently working with software development or have worked with it in the past, and who have hands-on experience with both JavaScript and TypeScript programming.

The survey contains ten questions and takes about 15 minutes to complete. The survey is answered anonymously and the identity of the respondent cannot be traced back to the answers. It is possible to give the name of the company you are representing as an optional answer in the survey, but that information is not published in any form as a part of the thesis, and is only used to possibly group results together during the analysis of the results.

By responding to this survey you will contribute to scientific research and will be able to influence the results of the survey. You may submit your aswers in either Finnish or English.

Many thanks in advance!

C Kyselyn kysymykset

1. What do you consider to be the most useful features of TypeScript, compared to plain JavaScript?

2. What do you consider to be the most significant drawbacks of TypeScript, compared to plain JavaScript?

3. Has the use of TypeScript had any effects on the planning phase of your software development process? If so, what kind of effects?

4. What kind of effects has the use of TypeScript had on the development phase of your software projects?

5. In what way has the use of TypeScript affected your test writing policies, compared to plain JavaScript?

6. What kind of effects has the use of TypeScript had on the maintenance phase of your software development process?

7. Have you used TypeScript, but then decided not to continue using it anymore. What were your reasons for the decision?

8. If you gave an answer to the previous question and have decided to not use TypeScript anymore, have you chosen some other alternative to TypeScript instead? In that case what alternative?

9. How likely would you recommend TypeScript to your friends or coworkers? (0 = not at all likely, 10 = very likely)

10. What company do you work for? (optional, given information is treated confidentially and the answer is only used for grouping answers)